“这是本很棒的书，对初学者来说更是如此。最重要的是Quigley给出的范例之多，涵盖面之广，使得这本书能包含多本书的内容。因此，只推荐这一本书就足够了。”

——Jim A. Lola



UNIX Shells by Example Fourth Edition

UNIX shell 范例精解（第4版）

(美) Ellie Quigley 著

李化 张国强 译

全面涵盖

Linux!

随书附带光盘

PEARSON Education

清华大学出版社

UNIX shell 范例精解

(第 4 版)

(美) Ellie Quigley 著
李化 张国强 译

清华大学出版社
北 京

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java一览无余：**Java**视频教程 | **Java SE** | **Java EE**

.Net技术精品资料下载汇总：**ASP.NET**篇

.Net技术精品资料下载汇总：**C#**语言篇

.Net技术精品资料下载汇总：**VB.NET**篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书**+**视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI脚本语言编程学习资源下载地址大全

Python语言编程学习资料**(**电子书**+**视频教程**)**下载汇总

最新最全**Ruby**、**Ruby on Rails**精品电子书等学习资料下载

数据库管理系统**(DBMS)**精品学习资源汇总：**MySQL**篇 | **SQL Server**篇 | **Oracle**篇

平面设计优秀资源学习下载 | **Flash**优秀资源学习下载 | **3D**动画优秀资源学习下载

最强**HTML/xHTML**、**CSS**精品学习资料下载汇总

最新**JavaScript**、**Ajax**典藏级学习资料下载分类汇总

网络最强**PHP**开发工具**+**电子书**+**视频教程等资料下载汇总

UML学习电子资下载汇总 软件设计与开发人员必备

经典**LinuxCBT**视频教程系列 **Linux**快速学习视频教程一帖通

天罗地网：精品**Linux**学习资料大收集**(**电子书**+**视频教程**) Linux**参考资源大系

Linux系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX操作系统精品学习资料<电子书**+**视频>分类总汇

FreeBSD/OpenBSD/NetBSD精品学习资源索引 含书籍**+**视频

Solaris/OpenSolaris电子书、视频等精华资料下载索引



图书在版编目(CIP)数据

UNIX shell 范例精解(第 4 版)/(美)奎格莉(Quigley, E.)著；李化，张国强 译. —北京：清华大学出版社，2007.5

书名原文：UNIX Shells by Example, Fourth Edition

ISBN 978-7-302-14589-9

Ⅰ. U… Ⅱ. ①奎…②李…③张… Ⅲ. UNIX 操作系统—程序设计 Ⅳ. TP316.81

中国版本图书馆 CIP 数据核字(2007)第 016394 号

责任编辑：王 军 王 婷
装帧设计：孔祥丰
责任校对：成凤进
责任印制：孟凡玉
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编**：100084
c-service@tup.tsinghua.edu.cn
社 总 机：010-62770175　　**邮购热线**：010-62786544
投稿咨询：010-62772015　　**客户服务**：010-62776969
印 刷 者：清华大学印刷厂
装 订 者：三河市金元印装有限公司
经　　销：全国新华书店
开　　本：185×260　**印　张**：57.75　**字　数**：1346 千字
附光盘 1 张
版　　次：2007 年 5 月第 1 版　　**印　　次**：2008 年 3 月第 2 次印刷
定　　价：118.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770177 转 3103　　产品编号：019262-01

译者序

作为一名经常使用UNIX/Linux系统进行软件开发的研发人员，我们对shell的功能和趣味性深有体会。然而国内本科教学，大都不把某个工具作为一门课程，对于shell也不例外。因此就造成了这样一种局面：国内开发人员阅读UNIX脚本的能力非常差，使用shell编程的人不多，也很少有人能够直接修改UNIX系统的启动配置。

2002年，我们有幸访问了英国某软件公司。当时需要将一个编译程序的目标码从Big-Endian计算机移植到环境基本相同的Little-Endian计算机上运行。该公司一名普通的程序员仅键入了几十行代码，创建了一个shell脚本，通过执行该脚本，就将原来在Big-Endian上的可执行程序转换成了在Little-Endian机器上的可执行程序。在感叹该程序员对机器指令与编译系统非常熟悉的同时，我们也见识到了shell脚本功能的强大。

shell编程通常不作为开发正式软件的方法，但是在安装、启动、格式转换、查找等方面作用显著，是提高工作效率的利器。

本书作为*UNIX Shells by Example*的第4版，主要增加了shell编程快速入门、调试shell脚本以及shell用于系统管理等方面的内容。作者作为有着21年教学经验的讲师，在组织书籍的结构方面，非常注重实用但不面面俱到，选择了那些基本的知识点进行讲解。对于每个知识点，作者先进行了简明扼要的说明，然后给出了若干组精心设计的范例，从而指导读者在实践中学习UNIX/Linux上所有主流shell的理论知识。

最后，欢迎各位读者对本书提供反馈意见。我们希望读者能从本书中受益，也希望通过读者的反馈意见来了解自己的不足，以求在今后的译作中更多更切实际地考虑读者的需要。请将您的反馈信息发送至wkservice@tup.tsinghua.edu.cn，我们将不胜感激。

译者

2006年12月



前　言

shell 游戏充满了乐趣。编写本书的目的就是使您的学习过程变得有趣而又充满收获。本书的第 1 版推出后，很多读者来信说，他们从我的书中得到了帮助，认识到 shell 编程根本就不难！范例让 shell 编程容易而有趣。正是因为您的肯定，Prentice Hall 才邀请我编写这本书的第 4 版。除了一些更新的内容之外，我在这个版本中增加了 3 章全新的内容。因为 Linux 在最近几年来的快速普及，我们在新书中对 Linux 中的多个 GNU 工具进行了全面的介绍，并对 UNIX/Linux shell 中的各项特征进行了详细阐述。

本版新增加的内容包括第 2 章、第 15 章和第 16 章。第 2 章“shell 编程快速入门”引导程序员尽快掌握 shell 编程的结构，并了解 shell 编程与其他语言编程的差别。第 15 章“调试 shell 脚本”给出了一些错误消息的范例，并告诉您导致错误的原因，以及如何修正错误。第 16 章“系统管理员与 shell”将展示系统管理员在从系统启动到关机的过程中如何使用 shell。

本书是我 21 年教学生涯的顶点，这些年来，我针对各种 shell 和程序员常用的 UNIX/Linux 工具设计了多门课程。我为这些课程编写的讲义被用于加州大学圣克鲁兹分校和戴维斯分校的 UNIX 教学、SUN 公司的培训。还被 Apple 公司、Xilinx 公司、美国国家半导体公司、LSI Logic 公司、DeAnZa 大学以及全球众多厂商采用。根据客户的需求，通常每次只讲授一种 shell，而不是一次讲授全部的 shell。为了满足众多客户的需要，我为每种 shell 和工具单独编写了培训教材。

无论是在讲授“grep、sed 和 awk”，“系统管理员 Bourne shell 教程”，“交互式的 korn shell”，还是“bash 编程”时，总有学生会问：“有没有一本书能够涵盖了所有的 shell 以及 grep、sed 和 awk 这些重要的工具？”，“awk 与 gawk 之间有什么区别？”，“某个工具能否在 Linux 系统上工作，还是仅仅适用于 Solaris 系统？”，“我是否应该拥有一本 awk 的书，或者要买一本关于 grep 和 sed 的书籍？”，“是否有一本书能够真正地覆盖所有这些内容？”，“我不希望为了成为一名 shell 程序员而购买三四本书”。

遇到这类问题时，我可以向学生们推荐一大堆好书，但是这些书籍只是单独讲述某个主题。也有一些 UNIX 参考书尝试覆盖所有的内容，但都只做蜻蜓点水式的介绍，学生们需要的却是详细的讲解。学生们希望有一本书能够包含他们需要的全部内容：各种工具、正则表达式、主流 shell、引用规则、各种 shell 的比较、练习等全都容纳在一本书中。本书就是这样的一本书。



编写这本书同时，我也在思考如何以相同的布局来教授课程和组织章节。在 shell 编程课程中，第一个主题不外乎介绍什么是 shell，它是如何工作的。然后讲述如 grep、sed 和 awk 等最重要的工具。在学习各种 shell 时，首先介绍的是它作为一个交互式程序，所有的事情都可以在它的命令行中完成。然后介绍了它作为一种编程语言的编程结构，并在 shell 脚本中进行说明(作为编程语言，C shell 与 TC shell 几乎完全相同，所以在描述它们的交互式应用时设立了单独的章节，而在讨论编程结构时仅用了一章的内容)。

在实际应用中，编写脚本是一回事，调试脚本则是另一回事。我在 shell 方面工作多年，因此在 bug 发生前我就能在程序中把它们找出来。但实际上这些 bug 是难以预料的，除非您已经看惯了错误信息并能理解它们的含义。因此，我针对调试中出现的问题增加了一章的内容以帮助您理解常见的错误提示信息及其含义，以及如何修正错误。由于不同版本的 shell 诊断信息可能不相同的，所以书中给出了每种 shell 常见的错误信息以及导致该错误的原因。

由于许多学生选择学习 shell 课程并想借此为学习系统管理铺路。于是，我的同事 Susan Barr，负责讲授系统管理与 shell 编程课程的讲师，也将自己的知识拿出来共享，也就有了系统管理员如何使用 shell 的这一章的内容(第 16 章“系统管理员与 shell”)，这里致以谢意。

我时常发现，简单的例子更容易理解，用一个小范例，然后附上输出，再对程序中的每一行进行解释，这样每个概念都能立即被掌握。对从我的第一本书 *Perl by Example* 中学习 Perl 编程，或从 *JavaScript™ by Example* 中学习 JavaScript，以及从 *UNIX® Shells by Example* 中学习编写 shell 程序的读者来说，这种方法已被证明是十分有效的。

本书另外一个有助于理解的特点是对 5 种 shell 的讨论是平行的。例如，某个时刻您正在一种 shell 上工作，但您希望看到重定向在另外一种 shell 上的情形，那么您将发现在每种 shell 的独立章节中，都有一个针对该主题的相应讨论。

当需要详细了解某个特定的命令如何工作时，您会因需在几本书或 man 手册中不停地翻来翻去而感到头疼。为节省时间，附录 A 包含了 UNIX 和 Linux 有用的命令列表，它们的语法以及定义。对常用命令还提供了范例和说明。

附录 B 中的对照表将帮助您理清不同 shell 之间的差别，特别是在将脚本从一种版本的 shell 移植到另外一种版本的 shell 上时，这一点显得更加重要。如果只想知道某种结构是如何工作的，那么也可以将它作为快速参考。

阅读本书，您将发现它是一本宝贵的指南和参考手册。本书的目标是通过范例讲解，将概念简化以使您获得乐趣并节省时间。因为这本书包含我在课堂上讲授的全部内容，所以我确信您将在短时间内成为一名高效率的 shell 程序员。您所要做的就是坐下来，翻开这本书，尽情享受 shell 游戏带来的乐趣。

Ellie Quigley(lequig@aol.com)

目　录



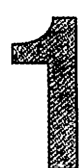


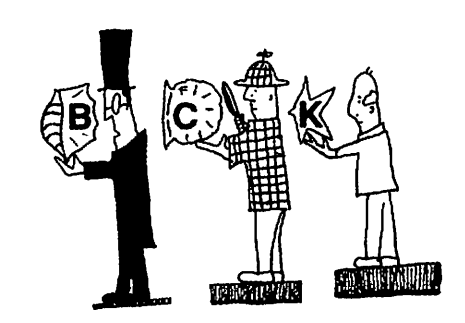

TopSage.com





chapter





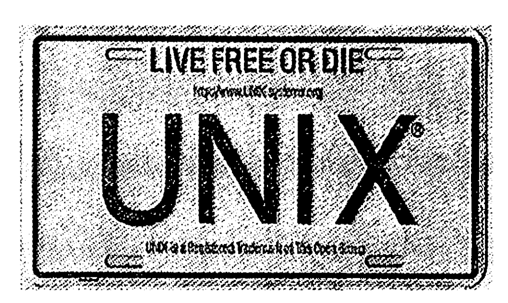
UNIX / Linux shell 简介

1.1 UNIX 与 Linux 及其历史

在学习 shell 时，我们发现 shell 通常与 UNIX/Linux 操作系统的不同版本相关联。例如，Bourne 和 Korn shell 经常与 AT&T UNIX 关联，C shell 与 Berkeley UNIX 关联，而 Bash shell 与 Linux 关联。在详细讨论 shell 之前，我们首先简单概述 shell 所驻留的操作系统的背景知识。

1.1.1 UNIX 简介

UNIX 是一个多用户、多任务的操作系统。最初由 AT&T 贝尔实验室的 Ken Thompson 于 1969 年开发成功。UNIX 当初设计的目标是允许大量程序员同时访问计算机，共享它的资源。它非常简单但是功能强大、通用并且可移植。可以运行在从微机到超级小型计算机以及大型机上。



UNIX 系统的心脏是内核：一个系统引导时加载的程序。内核用于与硬件设备打交道，调度任务，管理内存和辅存。正是由于 UNIX 系统这种精炼特性，众多小而简单的工具和实用程序被开发出来。因为这些工具(命令)能够很容易地组合起来执行多种大型的任务，所以 UNIX 迅速流行起来。其中最重要的工具之一就是 shell，一个让用户能够与操作系统沟通的程序。本书将剖析当今最主流 shell 的特性。

最初 UNIX 被科学研究机构和大学采用，其费用微不足道。后来慢慢扩展到计算机公司、政府机构和制造业领域。1973 年，美国国防部高级研究计划署(Defense Advanced Research Projects Agency，DARPA)启动一项计划，研究使用 UNIX 将跨越多个网络的计算机透明地连接在一起的方式。这个计划和从该研究中形成的网络系统，导致了 Internet 的

诞生！

在 20 世纪 70 年代后期，许多在大学期间接触并体验过 UNIX 的学生投身工业界并要求工业界向 UNIX 转换，声称它是最适合复杂编程环境的操作系统。很快大量或大或小的厂家，开始开发他们自己的 UNIX 版本，在自己的计算机体系结构上对它进行优化，以期占领市场。最著名的两个 UNIX 版本是 AT&T 的 System V 和 BSD UNIX。后者源于 AT&T 版本，由加州大学伯克力分校于 20 世纪 80 年代早期开发成功。

面对如此众多版本的 UNIX (有一个图表列出了 80 多个 UNIX 版本，访问 www.ugu.com/sui/ugu/show?ugu.flavors)，如果不花费时间和精力考虑兼容问题，则在一个系统上能够正常运行的应用程序和工具可能无法在另一个系统上工作。由于缺乏统一的标准，许多厂家放弃了 UNIX 转而使用比较古老的非 UNIX 专用系统，如 VMS，它们被证明是更加一致和可靠的。

统一 UNIX 标准的时候到了。一些厂商发起并启用了"开放式系统(open system)"的概念。以此为基础参与的厂商在遵循某个标准与规范上达成了一致。UNIX 被选为建立新概念的基础，并成立了 X/Open 公司负责定义开放操作系统平台，许多组织开始使用 X/Open 作为系统定义的基础。X/Open 现在是开放组(The Open Group)的一部分，还会继续开发单一的 UNIX 规范。

在 1993 年初，AT&T 将它的 UNIX 系统实验室出售给了 Novell。1995 年 Novell 将它的 UNIX 商标权和规范(后来变成了单一 UNIX 规范)转让给 The Open Group，将 UNIX 系统源代码卖给了 SCO。当今有很多公司都在出售基于 UNIX 的系统，包括 Sun Microsystems 的 Solaris，HP-UX 和来自 Hewlett-Packard 的 Tru64 UNIX 以及来自 IBM 的 AIX。除此之外，还有许多免费的 UNIX 和与 UNIX 兼容的工具，如 Linux、FreeBSD 和 NetBSD。

1.1.2 为什么选择 Linux



1991 年，芬兰的一个大学生 Linus Torvalds 在芬兰的赫尔辛基大学开发了一个与 UNIX 兼容的操作系统内核，它被设计成为 PC 上的 UNIX。尽管 Linux 模仿 UNIX System V 和 BSD UNIX，但它并未采用有版权的源代码，而是由一群来自世界各地的非正式结盟的开发人员通过互联网独立开发的。

对很多人来说，Linux 提供了 UNIX 和 Windows 操作系统之外的另一种选择。Linux 文化通过赞助联盟、大会、expos 软件、新闻组和出版物已经得到了很大的发展，它引领着一场挑战 PC 世界中 Windows 统治地位的新革命。在众多系统程序员和开发人员的帮助下，Linux 成为了当今与 UNIX 兼容的独挡一面的操作系统，拥有超过两千万的用户。当前功能最全面的内核是 2.6 版本(发布于 2003 年 11 月)，并且还在继续进行开发。有很多商业或非商业组织，如 Red Hat、Slackware、Mandrake、Turbo 和 SuSE Linux，发行了各种增强了操作系统内核功能的 Linux 发行版。

您可能已经注意到总有一个企鹅与 Linux 联系在一起。企鹅是 Linux 官方吉祥物，名

为 Tux，它是由 Linus Torvalds 选择的，以此反映他与 Linux 操作系统的关系。

免费软件基金会 1992 年，自由软件基金会将它的 GNU(GNU's Not UNIX)软件加入到 Linux 内核中以使其成为一个完整的操作系统，同时将 Linux 源代码置于它的通用公共许可证(General Public License，GPL)之下，从而使任何人都可以得到它。自由软件基金会提供了数以百计的 GNU 工具，包括对标准 UNIX Bourne shell 的改进。

GNU 工具，如 grep、sed 和 gawk，与 UNIX 上和它们同名的工具类似，但它们已改进且与 POSIX[①]兼容。在安装 Linux 时，会用到 GNU shell 和工具，而非标准的 UNIX shell 及工具。如果你用的是传统的 UNIX 系统，例如 Sun 公司的 Solaris 5.9，也可以使用许多这类工具，包括 GNU shell。

1.2 shell 的定义与功能

shell 是一种特殊的程序。如图 1-1 所示，它是用户与 UNIX/Linux 系统"心脏"(一个称作内核的程序)之间的接口。内核在系统引导时载入内存，管理系统直至关机。它创建和控制进程，管理内存、文件系统和通信等。内核以外的所有其他程序(包括 shell 程序)都保存在磁盘上。内核将这些程序加载到内存中运行，并在它们终止后清理系统。shell 是一个工具程序，在用户登录后系统启动。它解释并运行由命令行或脚本文件输入的命令，从而实现用户与内核间的交互。

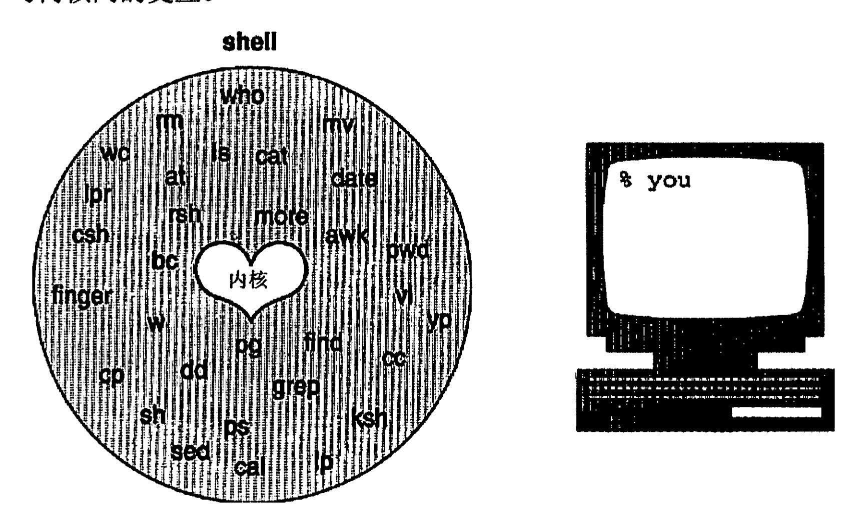


图 1-1 内核、shell 以及用户界面

当用户登录成功，系统会启动一个交互式的 shell 来提示用户输入指令。用户键入命令后，shell 开始执行任务：先解析命令行，再处理通配符、重定向、管道和作业控制，然后查找命令，找到后开始执行。UNIX/Linux 初学者大都通过交互方式使用 shell，即在命令提示符后逐条输入和执行命令。

① shell 功能需求在 POSIX(Portable Operating System Interface，可移植操作系统接口)标准中定义，也就是 POSIX 1003.2。



如果总要键入一组大致相同的命令，自然会希望将这些工作自动化。把命令写到一个文件中，即脚本文件，然后执行这个文件就可以实现这一想法。shell 脚本的作用跟批处理文件的很相似，都是把一组 UNIX/Linux 命令输入文件，然后执行该文件。更复杂的脚本还包括用于实现判断、循环、文件测试等功能的程序结构。编写脚本不仅要掌握编程结构和编程技巧，还需要对 UNIX/Linux 工具集及其运行机理有较深的理解。有一些工具，如 grep、sed 和 awk，在处理命令输出和文件时功能很强大。熟悉了这些工具和所用 shell 的程序结构后，就可以编写有用的脚本了。当用户从脚本中执行命令时，shell 被视作一种编程语言。

1.2.1 UNIX shell

UNIX 系统大都支持 3 种主流的 shell，它们是 Bourne shell(也称为 AT&T shell)、C shell(也称为 Berkeley shell)和 Korn shell(Bourne shell 的一个扩展集)。交互式运行时，这 3 种 shell 非常相似，但作为脚本语言，它们在语法和效率上有一定的差别。

Bourne shell 是标准的 UNIX shell，用于系统管理。大部分系统管理脚本(如 rc start 和 stop 脚本、shutdown)都是 Bourne shell 脚本。管理员以 root 身份运行的通常是 Bourne shell。Bourne shell 是 AT&T 开发的，以简练、紧凑和快速著称。默认的 Bourne shell 命令提示符是美元符号($)。

C shell 由美国加州大学 Berkeley 分校开发，增加了很多新的功能，比如命令行历史、别名、嵌入算术运算、文件名自动补全和作业控制。交互式用户喜欢 C shell，但管理员们更喜欢用 Bourne shell 编写脚本。因为同样的脚本，用 Bourne shell 编写更简单，运行更快。默认的 C shell 命令提示符是百分号(%)。

Korn shell 是 AT&T 的 David Korn 开发的，它是 Bourne shell 的一个扩展集。在增强改进 C shell 的基础上，Korn shell 添加了更多功能。Korn shell 的功能特点包括：可编辑的命令历史、别名、函数、正则表达式通配符、嵌入算术运算、作业控制、协同处理以及特殊的调试功能。Korn shell 几乎完全向上兼容 Bourne shell，所以老的 Bourne shell 程序在 Korn shell 中运行良好。Novell 公司开发了一个 Korn shell 的新版本(以前的版本为 ksh93)，支持在桌面上进行 X Windows 编程。dtksh.Dtksh 是大多数 UNIX 系统硬件厂商所支持的通用桌面环境(Common Desktop Environment，CDE)的标准部件，Korn shell 的公共域版本(packages.debian.org/stable/shells/pdksh)称为 pdksh，这种版本也在包括 Linux 在内的多种平台上可用。用于 Windows 的 Korn shell 可以在 www.wipro.com/uwin 上找到。默认的 Korn shell 提示符是美元符号($)。

1.2.2 Linux 的 shell

用户安装完 Linux 之后，就可以访问到 GNU 的 shell 和工具(非标准 UNIX 的 shell 和工具)。Linux 上默认的 shell 是 GNU bash(Bourne Again shell)，这是一种增强的 Bourne shell，其扩展的特性不仅表现在编程级别上，也表现在交互使用时，用户可以对自己的工作环境进行裁剪，建立快捷键以提高工作效率。bash 是当前 UNIX 和 Linux 用户使用得最为普遍的 shell，可以通过 www.gnu.org/software/bash/bash.html 下载。默认的 Bash 提示符为美元符号($)。

Linux 用户常用的另一个 shell 是 TC shell。TC shell 是 UNIX C shell 的一个兼容分支，但是新增了许多附加功能[②]，具体请访问 www.tcsh.org/MostRecentRelease。默认的 C shell 提示符大于符号(>)。

Z shell 也是 Linux 一种 shell，它结合了 Bourned Again shell、TC shell 和 Korn shell 的许多功能，具体请访问 sourceforge.net/projects/zsh/。

还有一种称为 Public Domain Korn shell (pdksh)的 shell，这种 shell 是 Korn shell 的克隆版本，使用这种 shell 需要向 AT&T 付费，Public Domain Korn shell 默认的提示符是美元符号($)。

要知道在自己所使用的 Linux 有哪些版本的 shell，可以查看/etc/shell 目录下的文件。如范例 1-1 所示。

范例 1-1

```
$ cat /etc/shell
/bin/bash
/bin/sh
/bin/ash
/bin/bsh
/bin/tcsh
/bin/csh
/bin/ksh
/bin/zsh
```

说明

Linux 上的/etc/shell 文件中包含所有在当前 Linux 环境中可用的 shell 程序。常用的版本包括 bash(Bourne Again shell)、tcsh(TC shell)和 ksh(Korn shell)。

什么是 POSIX 为了给不同的操作系统及其程序提供软件标准，人们提出了 POSIX 标准(也称为操作系统标准)，POSIX 标准的参与者包括美国电子与电器工程师协会(IEEE)和国际标准化组织(ISO)。其目标是在不同平台之间提供应用程序的可移植性标准，以便提供一个类 UNIX 的计算机环境。这样，在一个机器上编写的新软件，可以在另一个配有不同硬件的机器上编译和运行。例如，在 BSD UNIX 机器上编写的程序同样可以在 Solaris、Linux 和 HP-UX 机器上运行。1988 年，第一个 POSIX 标准，POSIX 1003.1 问世。其目的是为了提供一个 C 语言的标准。1992 年，POSIX 组为了开发可移植的 shell 脚本，建立了 shell 和工具软件的标准，称为 IEEE 1003.2 POSIX shell 标准及通用工具程序。尽管这些标准并没有严格的约束，大多数 UNIX 开发商仍尽量与 POSIX 标准兼容。在讨论 shell 以及通用的 UNIX 工具软件时，“POSIX 兼容性”这个术语指的是在编写新的工具软件和对现有的工具软件进行扩展时，力求与 POSIX 委员会描述的标准兼容。例如，Bourne Again shell 是一种几乎 100%兼容的程序，而 gawk 是一个仅在严格 POSIX 模式下运行的用户软件。

② 尽管 bash(Bourne Again shell)和 tcsh(TC shell)这两个 shell 经常被称作“Linux 的”shell，但它们其实是免费软件，并且可以在任何 UNIX 系统上编译。实际上，Solaris 8+(Sun 的 UNIX 操作系统)上现在已经捆绑了这两种 shell。



1.3 shell 的历史

第一个重要的标准 UNIX shell 于 1979 年末在 V7(AT&T 的第 7 版)UNIX 上推出，并以作者 Stephen Bourne 的名字命名。作为编程语言，Bourne shell 基于另一种叫做 Algol 的语言。Bourne shell 当时主要用于系统管理任务的自动化，Bourne shell 以简单和高速而受欢迎，却缺少了很多用于交互的功能，如命令历史、别名和作业控制。

C shell 由加州大学 Berkeley 分校于 20 世纪 70 年代末开发，作为 2BSD UNIX 系统的一部分发布。它的主要开发者是 Bill Joy。C shell 提供了很多标准的 Bourne shell 不具备的功能。C shell 基于 C 语言，作为编程语言使用时，语法也类似于 C。C shell 也提供了增强交互使用的功能，如命令行历史、别名和作业控制。由于是为大型机设计并增加了很多新功能，C shell 在小型机器上运行可能比较慢。而且，即使在大型机上，它的速度也不如 Bourne shell。

Bourne shell 和 C shell 共存使 UNIX 用户有了选择，也导致了人们对哪个 shell 更好的争论。20 世纪 80 年代中期，AT&T 的 David Korn 推出了 Korn shell。Korn shell 于 1986 年发布，并在 1988 年 UNIX 的 SVR4 版本发布时正式成为它的一部分。Korn shell 其实是 Bourne shell 的一个扩展集，它不仅能运行于 UNIX 系统，还能在 OS/2、VMS 和 DOS 上运行。Korn shell 提供了对 Bourne shell 的向上兼容性，加入了许多 C shell 中受欢迎的功能，而且快速和高效。Korn shell 经历了许多版本，虽然 1993 版正逐渐流行，目前用得最广泛的还是 1988 版。Linux 用户可能会发现自己正在使用 Korn shell 的免费版本，叫作 Public Domain Korn shell，简称 pdksh。pdksh 是 1988 版 Korn shell 的克隆版。pdksh 是免费和可移植的。对它的改进正在进行中，以使其能够完全兼容 Korn shell，并且符合 POSIX 标准。此外还有 Z shell(zsh)，这也是一个 Korn shell 的克隆版，集成了 TC shell 的一些功能。Z shell 的作者是 Paul Falsted，可以从很多网站免费获取。

随着 Linux 的发展，Bourne Again shell(bash)开始流行起来。自由软件基金会的 Brian Fox 取得 GNU 版权许可后于 1988 年开发出 bash。bash 是 Linux 操作系统上默认的 shell。它的设计符合了 IEEE POSIX P1003.2/ISO 9945.2 shell 和工具标准。在交互和编程两方面，bash 都提供了很多 Bourne shell 没有的功能(但 Bourne shell 脚本无需修改还能在 bash 下运行)。bash 还结合了 C shell 和 Korn shell 最有用的功能，它真的很棒。bash 对 Bourne shell 的改进包括：命令行历史与编辑、目录栈、作业控制、函数、别名、数组、整数运算(底数可以是 2~64)，以及 Korn shell 的一些功能，如扩展的元字符，用于生成菜单的 select 循环和 let 命令等。

TC shell 是 C shell 的扩展版本，且具有完全兼容性。新增的功能包括：命令行编辑(emacs 和 vi)、历史清单的滚动、高级的文件名功能、变量和命令补全、拼写纠错、作业调度、账户自动上锁和注销、历史清单中增加时间戳等。新增的功能确实很多。人们经常会问“TC shell 中的 T 到底代表什么含义呢？”这就要涉及到一段历史。1976 年，DEC 发布了一种新的虚拟内存操作系统——TOPS-20，这种操作系统基于 TENEX，可以被美国国内的多个研究人员同时使用。TOPS-20 最显著的特点是“遗忘识别”，也称为“命令补全”，用户可以借助 Esc 键获取大多数的命令或助记符，从而使得系统能正常运行。TC shell 的创建者

受到 TENEX/TOPS-20 的这个功能以及其他功能的影响，开发了 csh 的一种版本，并模仿 TENEX 的名称，将这种 shell 称为 TENEX C shell，简称为 TC shell、tc-shell 或 tcsh。关于 tcsh 的更多信息，请访问 info.astrian.net/doc/tcsh/copyright。

1.3.1 shell 的作用

shell 的一项主要功能是在交互方式下解释从命令行输入的命令。shell 解析命令行，将其分解为词(也称为 token)，词之间由空白分隔，空白由制表符、空格键或换行组成。如果词中有特殊的元字符，shell 会对其进行替换。shell 处理文件 I/O 和后台进程。对命令行的处理结束后，shell 搜索命令并开始运行它。

shell 的另一项重要功能是定制用户环境，这通常在 shell 的初始化文件中完成。初始化文件中有很多定义，包括设置终端键和窗口属性，设置用来定义搜索路径、权限、提示符和终端类型的变量，设置特定应用程序所需的变量，如窗口、字处理程序和编程语言的库等。Korn/Bash shell 和 C/TC shell 还提供了更多的定制功能：历史添加、别名、设置内置变量防止用户破坏文件或无意中退出，通知用户作业完成。

shell 还能用作解释性的编程语言。shell 程序(也称为 shell 脚本)由文件中的一列命令组成。shell 程序用编辑器生成(也可以在命令行上直接输入脚本)。它们由 UNIX 命令组成，命令之间插入了一些基本的程序结构，如变量赋值、条件测试和循环。shell 脚本不需要编译。shell 会逐行解释脚本，就好像它是从键盘输入一样。shell 负责解释命令，因此，用户需要了解可用的命令有哪些。附录 A 中列出了一些有用的命令。

1.3.2 shell 的职责

shell 负责确保用户在命令提示符后输入的命令被正确执行。其职责包括：

(1) 读取输入并解析命令行
(2) 替换特殊字符，比如通配符和历史命令符
(3) 设置管道、重定向和后台处理
(4) 处理信号
(5) 程序执行的相关设置

与具体的 shell 相关的内容将在本书中详细讨论。

1.4 系统启动与登录 shell

系统启动时运行的第一个进程是 init。每个进程都有一个称为 PID 的进程标识号。init 是第一个进程，所以它的 PID 是 1。init 进程初始化系统，启动另一个进程来打开终端线路并设置标准输入(stdin)，标准输出(stdout)和标准错误输出(stderr)，三者都与终端关联。标准输入通常来自键盘，标准输出和标准错误输出则显示在屏幕上。完成这些设置后，终端上就会出现登录提示。

系统会在用户键入用户名后提示输入口令。程序/bin/login 通过检查 passwd 文件的首个字段来确认用户的身份。如果所键入的用户名存在，它会运行一个密码程序来对所键入



的口令进行确认。口令验证通过后，login 程序设置初始环境。初始环境是一组定义工作环境的变量，这组变量将传给 shell。变量 HOME、SHELL、USER 和 LOGNAME 根据 passwd 文件中的信息进行赋值。HOME 被设为用户的主目录，SHELL 则被设为登录 shell 的名字，即 passwd 文件中的最后一列。USER 和 LOGNAME 被赋值为登录名。还设置了变量 search path，常用的工具程序可以在该变量指定的目录中找到。login 程序结束时执行它在 passwd 文件最后一列中找到的程序。这个程序通常是一个 shell。如果 passwd 文件最后一列是/bin/csh，执行的就是 C shell。如果是/bin/sh 或为空，则执行 Bourne shell。如果是/bin/ksh 或/bin/pdksh，则执行 Korn shell。被执行的 shell 称为登录 shell。

shell 启动后，先查找由系统管理员设置的系统级的初始化文件，然后在用户的主目录中查找是否存在对应的 shell 初始化文件。如果存在，就会执行这些文件。这些初始化文件用于进一步定制用户环境。在执行完这些初始化文件之后，就可以启动窗口界面的开发环境，如 CDE、Open Windows 或 Gnome。接着，将显示一个虚拟桌面，该桌面的显示基于配置、控制台以及显示 shell 提示符的伪终端，此时 shell 正处于等待输入状态。

1.4.1 解析命令行

在命令提示符后输入一条命令后，shell 会读入这个命令行并对命令行进行解析，将其分解为词，称作 token。token 之间用空格或制表符分隔，命令行以换行符结尾[③]。接下来，shell 检查第一个词是否为内置命令或磁盘上的可执行程序。如果是内置命令，shell 就在自己内部执行它。否则，shell 将在路径变量所指的目录中查找这个程序。如果找到了命令的程序，shell 就创建一个进程来执行它。之后，shell 进入睡眠(或等待)状态直至程序执行完毕。shell 会根据需要报告程序的退出状态。此时，屏幕上又会出现命令提示符，整个过程从头开始。命令行的处理顺序如下：

(1) 执行历史命令替代(视情况而定)
(2) 命令行被分解为 token(或称“词”)
(3) 更新历史命令(视情况而定)
(4) 引用的处理
(5) 别名替代和函数的定义(视情况而定)
(6) 设置重定向，后台进程和管道
(7) 执行变量替换(如$name，$user 等)
(8) 执行命令替换(如`date`代替 Today)
(9) 执行称为 globbing 的文件名替换(如 cat abc.??，rm *.c 等)
(10) 执行命令

1.4.2 命令类型

命令被执行时，可以是别名、函数、内置命令或磁盘上的一个可执行程序。别名是原有命令的缩写(昵称)，可用于 C、TC、bash 和 Korn shell。函数可用于Bourne shell(在 AT&T System V 的第 2 版中引入)、bash 和 Korn shell。函数是由像独立的例程一样的指令组织起

③ 将命令行分解成 token 的过程称为词法分析(lexical analysis)。

来的一组命令。别名和函数都在 shell 的内存空间中定义。内置命令是 shell 的内部程序，而可执行程序则在磁盘上。shell 用路径变量在磁盘上定位可执行程序。执行命令前，shell 需要创建一个子进程。程序定位和子进程创建都要花一定的时间。执行命令前，shell 按如下顺序判定命令类型[④]：

(1) 别名
(2) 关键字
(3) 函数
(4) 内置命令
(5) 可执行程序

举个例子，如果命令是 xyz，则 shell 先检查 xyz 是不是一个别名。如果不是，再检查它是不是内置命令或函数。如果都不是，那它肯定是磁盘上的一个可执行程序。接下来 shell 就要开始查找这个命令的路径以便执行。该过程的说明见图 1-2。

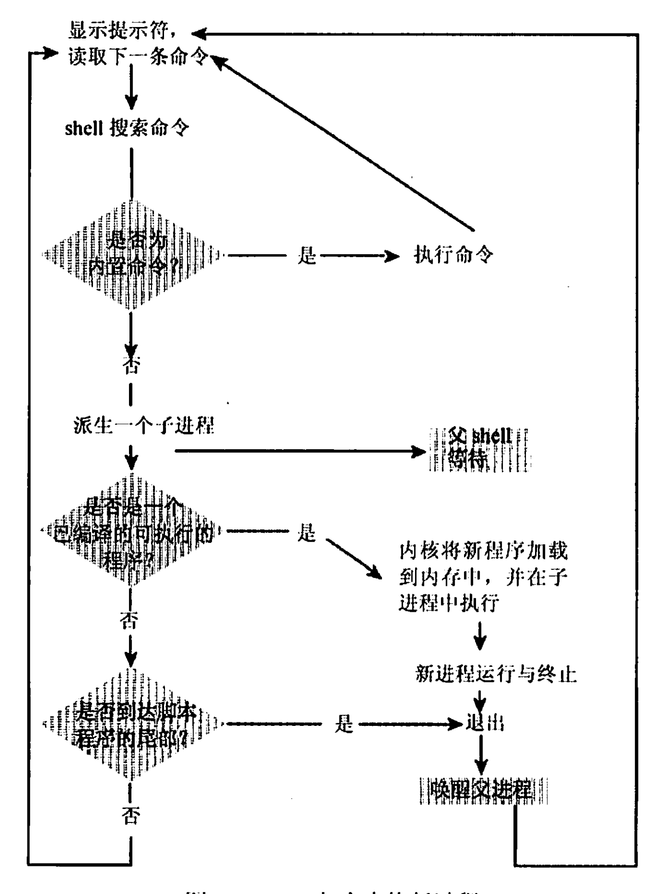


图 1-2 shell 与命令执行过程

④ 在 Bourne shell 和 Korn(88) shell 中，步骤 3 和步骤 4 是颠倒的。而对于 C 和 TC shell，则没有步骤 3。



1.5 进程与 shell

进程是正在运行的程序，可以用它唯一的 PID(进程标识，process identification)号来标识。内核负责控制和管理进程。进程由可执行程序、进程的数据和堆栈、程序指针和堆栈指针、寄存器以及程序运行时需要的所有信息组成。用户登录后，shell 这个特殊的程序将自行加载。shell 启动后，就生成了一个进程，并且属于某个进程组。进程组用组 PID 进行标识。任何时候只能有一个进程组拥有终端的控制权，即所谓的在前台运行。当用户登录系统后，shell 便控制了终端，等待用户在命令提示符后输入命令。

登录后，系统可能直接进入图形用户界面(GUI)，也可能启动进入带有 shell 提示符的终端。如果使用的是 Linux，shell 通常会创建另一个进程来启动 X Windows 系统。X Windows 启动之后，将执行窗口管理器进程(twm、fvwm 等)，并提供一个虚拟桌面[5]。然后用户可以通过弹出菜单，启动一些其他的进程，如 xterm(获取一个终端)、xman(提供用户手册)、emacs(启动一个本文编辑器)等。如果使用的是 UNIX，GUI 可能是 CDE、KDE 或 OpenWindows。一旦启动了 GUI，用户就获得了一个包含一些小窗口的虚拟桌面，这个虚拟桌面包含一个控制台和一些终端，每个终端都会显示一个 shell 提示符。

Linux/UNIX 内核运行着多个进程，并对其进行监控，为每一个进程分配一段 CPU 时间，通过这样的方式让用户感觉好像自己独占系统资源。

1.5.1 哪些进程正在运行？

ps 命令 ps 命令和它的众多选项能够以多种格式列出当前正在运行的进程。范例 1-2 显示的是在 Linux 系统上由用户运行的所有进程(关于 ps 和它的选项请参见附录 A)。

范例 1-2

```
$ ps aux  (BSD/Linux ps)  (use ps -ef for SVR4)
USER  PID  %CPU %MEM  SIZE  RSS  TTY STAT START  TIME  COMMAND
ellie 456  0.0  1.3   1268  840  1    S   13:23  0:00  -bash
ellie 476  0.0  1.0   1200  648  1    S   13:23  0:00  sh /usr/X11R6/bin/sta
ellie 478  0.0  1.0   2028  676  1    S   13:23  0:00  xinit /home/ellie/.xi
ellie 480  0.0  1.6   1852  1068 1    S   13:23  0:00  fvwm2
ellie 483  0.0  1.3   1660  856  1    S   13:23  0:00  /usr/X11R6/lib/X11/fv
ellie 484  0.0  1.3   1696  868  1    S   13:23  0:00  /usr/X11R6/lib/X11/fv
ellie 487  0.0  2.0   2348  1304 1    S   13:23  0:00  xclock -bg #c0c0c0 -p
ellie 488  0.0  1.1   1620  724  1    S   13:23  0:00  /usr/X11R6/lib/X11/fv
ellie 489  0.0  2.0   2364  1344 1    S   13:23  0:00  xload -nolabel -bg gr
ellie 495  0.0  1.3   1272  848  p0   S   13:24  0:00  -bash
ellie 797  0.0  0.7   852   484  p0   R   14:03  0:00  ps au
root  457  0.0  0.4   724   296  2    S   13:23  0:00  /sbin/mingetty tty2
root  458  0.0  0.4   724   296  3    S   13:23  0:00  /sbin/mingetty tty3
root  459  0.0  0.4   724   296  4    S   13:23  0:00  /sbin/mingetty tty4
root  460  0.0  0.4   724   296  5    S   13:23  0:00  /sbin/mingetty tty5
root  461  0.0  0.4   724   296  6    S   13:23  0:00  /sbin/mingetty tty6
```

[5] 在 Linux 上有很多的桌面环境，包括 Gnome、KDE、X 等。

```
root   479  0.0   4.5   12092 2896 1   S   13:23 0:01  X :0
root   494  0.0   2.5   2768  1632 1   S   13:24 0:00  nxterm -ls -sb -fn
```

pstree/ptree 命令　另一种查看进程运行情况及子进程的方法是使用 pstree(Linux 系统适用)命令或 ptree(Solaris 系统适用)命令。pstree 命令以树状的方式显示所有的进程，其根进程 init 将首先运行。如果指定了用户名，则用户进程就是树的根。如果一个进程派生了多个同名的进程，pstree 将同样的分支用方括号组合起来，并以进程的个数为前缀来命名。为了说明，范例 1-3 所示的是 httpd 服务器进程启动 10 个子进程时的情形(关于 pstree 的更多选项，参见附录 A)。

范例 1-3

```
pstree
init---4*[getty]
init-+-atd
     |-bash---startx---xinit-+-X
     |                       '-fvwm2-+-FvwmButtons
     |                               |-FvwmPager
     |                               '-FvwmTaskBar
     |-cardmgr
     |-crond
     |-gpm
     |-httpd---10*[httpd]
     |-ifup-ppp---pppd---chat
     |-inetd
     |-kerneld
     |-kflushd
     |-klogd
     |-kswapd
     |-lpd
     |-2*[md_thread]
     |-5*[mingetty]
             |-nmbd
     |-nxterm---bash---tcsh---pstree
     |-portmap
     |-sendmail
     |-smbd
     |-syslogd
     |-update
     |-xclock
     `-xload
```

1.5.2 系统调用

shell 可以派生其他的进程。事实上，如果用户在提示符后输入一个命令或者运行 shell 脚本，shell 会负责找到相应的命令(可能是内置命令、磁盘上的命令)并执行。这个过程是通过调用内核完成的，称为系统调用(system call)。系统调用是对内核服务的请求，并且是进程访问系统硬件的唯一方式。有大量的系统调用可以创建、执行和终止进程(当执行重定向、管道、命令替换和用户命令时，shell 通过内核提供其他的服务)。

有关 shell 通过系统调用派生进程的过程将在下一节进行描述，参见图 1-4(第 14 页)。



1.5.3 创建进程

系统调用 fork 在 UNIX 系统中，进程是通过系统调用 fork 创建的。系统调用 fork 创建调用进程的一个副本。新创建的这个进程称为**子进程(child)**，创建它的进程称为**父进程(parent)**。fork 调用完成后，子进程立即开始运行，最初阶段，父子进程共享 CPU。子进程还将得到父进程信息的副本，包括环境、打开的文件、真实且有效的用户标识、umask、当前工作目录和信号。

用户输入命令后，shell 开始解析命令行，判断第一个单词是内置命令还是磁盘上的可执行命令。如果是内置命令，就由 shell 来处理；如果是磁盘上的命令，shell 就调用 fork 来生成自己的一个副本(参见图 1-3)。子进程将搜索路径找到这个命令，并且设置用于重定向的文件描述符、管道、命令替换和后台处理。子 shell 运行时，父 shell 通常处于睡眠状态(参见下面介绍的系统调用 wait)。

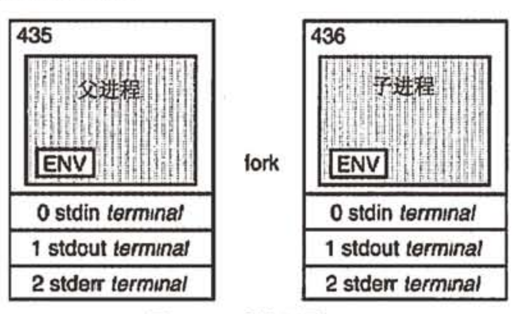


图 1-3 系统调用 fork

系统调用 wait 当子进程处理重定向、管道和后台处理等细节时，父 shell 进入睡眠状态。系统调用 wait 导致父进程挂起直至它的一个子进程终止。如果 wait 调用成功，它将返回死去子进程的 PID 和子进程的退出状态。如果父进程没有等待，子进程死亡时就进入僵尸(假死)状态，并且保持这个状态直至父进程调用 wait 或死亡[⑥]。如果父进程先于子进程死亡，则 init 进程会处理遗留的僵尸子进程。因此，系统调用 wait 不仅仅用于让父进程进入睡眠状态，还可用于保证进程正常终止。

系统调用 exec 在终端输入一条命令后，shell 通常会派生一个新的 shell 进程，即子进程。前面提过，子 shell 负责运行用户输入的命令。它通过系统调用 exec 实现这一点。记住，用户命令其实就是可执行程序。shell 在路径中查找新程序。如果找到了，shell 就以命令的名字作为参数调用系统调用 exec。内核把新程序加载到内存，替换掉调用它的 shell。于是，子 shell 便被这个新程序覆盖，新程序成为子进程并开始执行。虽然新进程有自己的局部变量，但是所有的环境变量、打开文件、信号和当前工作目录还是会保留。这个进程结束后就退出，然后唤醒父 shell。

系统调用 exit 新程序随时可以通过调用 exit 终止。子进程终止时，会向父进程发出一个信号(sigchild)并等待父进程接受它的退出状态。退出状态是一个 0~255 之间的数。退出状态为 0 说明程序执行成功，不为 0 则表示发生了某种错误。

⑥ 要清除僵尸进程，必须重启系统。

例如，如果在命令行键入命令 ls，父派生将派生一个子进程，然后开始睡眠。接下来，子 shell 将在自己的位置执行(覆盖)ls 程序。ls 程序于是替代子 shell，同时继承了它的所有环境变量、打开文件、用户信息和状态信息。这个新进程运行结束时将调用 exit 退出，父进程也将被唤醒。这时，显示屏上会出现命令提示符，shell 开始等待下一条命令。如果您有兴趣了解命令是如何退出的，每种 shell 都有一个特殊的内置变量，它包含上一条命令的退出状态(所有这些内容都会在 shell 所对应的独立章节中详细介绍)。请参考范例 1-4 中创建和终止进程的示例。

范例 1-4

```
(C/TC Shell)
1   % cp filex filey
    % echo $status
    0
2   % cp xyz
    Usage: cp [-ip] f1 f2; or: cp [-ipr] f1 ... fn d2
    % echo $status
    1

(Bourne, Bash, Korn Shells)
3   $ cp filex filey
    $ echo $?
    0
    $ cp xyz
    Usage: cp [-ip] f1 f2; or: cp [-ipr] f1 ... fn d2
    $ echo $?
    1
```

说明

1. 在 C shell 的命令行提示符下输入 cp(复制)命令。该命令把 filex 复制到 filey 后，程序退出，命令提示符再次出现。csh 的变量 status 包含 shell 执行的上一条命令的退出状态。如果退出状态为 0，则说明 cp 程序是成功退出，退出状态不为 0 说明程序发生了某种错误。

2. 输入 cp 命令时，用户没有提供所需的两个文件名：源文件和目标文件。于是 cp 程序向屏幕发送了一条错误消息，然后以状态 1 退出。这个状态数保存在 csh 的变量 status 中。0 以外的任何状态都表示程序运行出错。

3. Bourne、Bash 和 Korn shell 像上面两个例子中 C shell 那样执行了 cp 命令。唯一的区别是，Bourne shell 和 Korn shell 把退出状态保存在变量“?”而不是 status 中。

说明

1. 父 shell 通过系统调用 fork 创建自己的一个副本。这个副本称为子 shell。

2. 子 shell 有一个新的 PID，是父进程的副本。它将和父进程共享 CPU。

3. 内核把 grep 程序载入内存，并且执行(exec)它，替换掉子 shell。grep 程序继承了子 shell 的已打开文件和工作环境。

4. grep 程序退出，内核负责清理工作，父进程被唤醒。



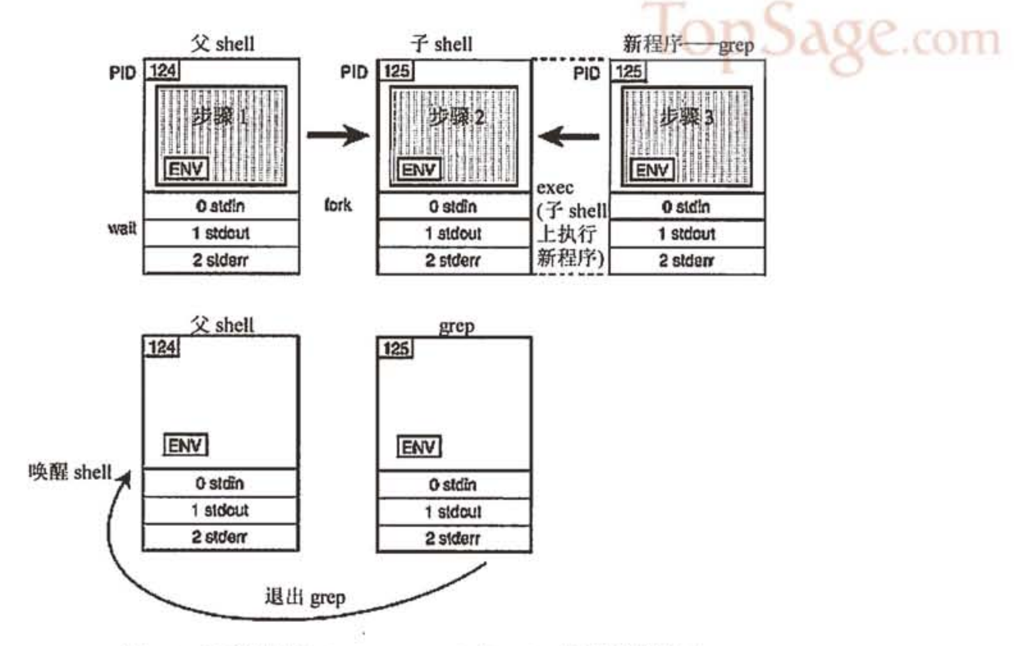


图 1-4　系统调用 fork、exec、wait 和 exit，具体说明如下

终止进程　进程可以通过 Ctrl+C 或 Crtl+\组合键，或者通过 kill 命令来终止。kill 命令可以终止后台作业，如果因为程序原因屏幕不再响应，也可以终止出现问题的进程。kill 命令是一个内置的 shell 命令，它通过 PID 来终止进程，如果使用作业控制的话，也可以通过作业号终止作业。查找 PID 号可以使用 ps 命令。参见范例 1-5。

范例 1-5

```
1    $ sleep 60&
2    $ ps
     PID   TTY      TIME CMD
     27628 pts/7    0:00 sleep
     27619 pts/7    0:00 bash
     27629 pts/7    0:00 ps
3    $ kill 27628
4    $ ps
     PID   TTY      TIME CMD
     27631 pts/7    .0:00 ps
     27619 pts/7    0:00 bash
     [1]+  Terminated             sleep 60
```

说明

1. sleep 命令不做实质性工作，仅仅暂停 60 秒。末尾的&符号设置程序在后台运行。

2. ps 命令显示当前用户运行的进程。

3. kill 命令是一条内置的 shell 命令。它以 PID 为参数终止相应的进程。输入的是 sleep 命令的 PID。

4. ps 命令显示 sleep 命令已终止。

1.6 环境与继承

用户进入系统后，shell 就开始运行，并且从启动它的程序/bin/login 那里继承了一大堆变量、I/O 流和进程特征。反过来，如果登录 shell 或某个父 shell 派生了一个子 shell，这个子 shell 也会从它的父 shell 那继承某些特征。启动子 shell 的原因有很多：为了进行后台处理、为了处理命令组或者为了执行脚本。子 shell 从父 shell 那里继承了一个环境。这个环境包括进程权限(谁拥有该进程)、工作目录、文件的创建掩码、特殊变量、打开文件和信号。

1.6.1 所有权

用户登录成功后，系统会给用户的 shell 一个身份，包括一个真实用户标识(UID)，一个或多个真实用户组标识(GID)以及一个有效用户标识(EUID)和有效用户组标识(EGID)。开始时，EUID 和 EGID 分别与 UID 和 GID 相同。这些 ID 号可以在 etc/passwd 文件中找到，它们被系统用来标识用户和用户组。EUID 和 EGID 决定了进程读、写或执行文件时的访问权限。如果进程的 EUID 与文件所有者的真实 UID 相同，则进程具有文件的所有者访问权限。如果进程的 EGID 和真实 GID 相同，则进程拥有所有者的组特权。

真实 UID 来自文件/etc/passwd，它是与用户的登录名相关联的一个正整数。真实 UID 是口令文件的第 3 个字段。在用户登录成功后，其真实 UID 将赋给登录 shell，之后所有从登录 shell 派生的进程都会继承它的权限。所有 UID 为 0 的进程都属于根用户(超级用户)，具有根用户权限。真实用户组标识，即 GID，是与用户登录名关联的一个用户组。它是/etc/passwd 文件的第 4 个字段。

可以使用 id 命令来查看这些值，参见范例 1-6。

范例 1-6

```
1  $ id
   uid=502(ellie) gid=502(ellie)
```

可以将 EUID 和 EGID 改为另一个所有者的 ID 号。可以通过把 EUID(或 EGID)改为另一个所有者的 ID 号而使您拥有其他用户的进程。把 EUID 或 EGID 改为另一个所有者的程序称为 setuid 或 setgid 程序。程序/etc/passwd 就是一个 setuid 程序的例子，它让普通用户拥有根用户的权限，从而能使普通用户修改口令，更新自己的 passwd 文件(这个文件只对根用户可访问)而无需通知系统管理员。由于用户 ID 是临时设为 root，所以用户只在 passwd 程序运行时临时地拥有最高权限。setuid 程序往往是安全漏洞的根源。shell 允许用户创建 setuid 脚本，而且 shell 本身也可以是 setuid 程序(参见第 16 章“系统管理员与 shell”)。

1.6.2 为文件创建掩码

创建文件时，会得到一组默认的权限。这组权限由创建文件的进程决定。子进程从它们的父进程那继承了一个默认掩码。用户可以通过两种方式改变 shell 的掩码：在命令提示符后输入命令 umask，或者在 shell 的初始化文件中设置它。umask 命令用于从当前掩码中除去某些权限。



一开始，掩码是 000，即默认情况下，目录的权限设为 777(rwxrwxrwx)，文件的权限则是 666(rw-rw-rw-)。大多数系统会通过程序/bin/login 或初始化文件/etc/profile 把 umask 的值设为 022。

目录和文件都会从它们的默认权限中减去 umask 的值，如下所示。

```
 777(目录)              666(文件)
-022(umask 的值)       -022(umask 的值)
--------------         ---------------
 755                       644
结果: drwxr-xr-x       -rw-r--r--
```

设置进程的 umask 后，进程所创建的所有目录和文件都将赋予新的默认权限。上例中，目录的所有者被赋予读、写和执行权限，所属的组则拥有读和执行权，其他用户拥有读和执行权。所有文件的所有者被赋予读写权，所属的组和其他用户则只有读权限。要改变单个目录或文件的权限，应该使用 chmod 命令。

1.6.3 修改权限与所有者

chmod 命令 chmod 命令用来改变文件和目录的访问权限。每一个 UNIX/Linux 文件都有一个控制其读、写和执行的权限集合。每个 UNIX/Linux 文件仅有一个所有者且只有所有者或超级用户才能使用 chmod 命令改变文件或目录的权限。一个用户组可以有多个成员，文件的所有者可以改变文件的组权限，让用户组享有特殊的权利。要查看文件的权限，可以在 shell 提示符下键入下面的命令：

```
ls -l filename
```

文件权限由 9 个二进制位组成，每三位为一组。第一组控制文件所有者的权限，第二组控制文件的组权限，最后一组控制的则是所有其他人的权限。文件的权限保存在它的 inode 的 mode 字段中。用户必须是文件所有者才能修改它的权限[⑦]。

表 1-1 给出了 8 种可能的数字组合，用于修改权限。

表 1-1 权限模式

十 进 制	二 进 制	权 限
0	000	none
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

chmod 的符号含义：r=read，w=write，x=execute，u=user，g=group，o=others，a=all。

⑦ 调用者的 EUID 必须与文件所有者的 UID 相同，或者所有者是超级用户。

范例 1-7

```
1   $ chmod 755 file
    $ ls -l file
    -rwxr-xr-x 1 ellie  0 Mar  7 12:52 file
2   $ chmod g+w file
    $ ls -l file
    -rwxrwxr-x  1 ellie  0 Mar 7 12:54 file
3   $ chmod go-rx file
    $ ls -l file
    -rwx-w---- 1 ellie  0 Mar 7 12:56 file
4   $ chmod a=r file
    $ ls -l file
    -r--r--r-- 1 ellie  0 Mar 7 12:59 file
```

说明

1. 第一个参数是八进制数 755。它设置了用户的 rwx 权限，组和其他用户的 r 和 w 权限。

2. chmod 的符号形式，为组增加了写权限。

3. chmod 的符号形式，取消了组和其他用户的读和执行权限。

4. chmod 的符号形式，只给所有用户读权限。等号使所有权限被重置为新值。

chown 命令　chown 命令将改变文件和目录的所有者及所在组。如果使用的是 Linux 或 UNIX，则只有 root 超级用户或文件所有者才能改变从属关系。要了解 chown 的用法及选项，请参考 UNIX 的手册，或者使用 chown --help(Linux)命令来获取帮助。参见范例 1-8 和范例 1-9。

范例 1-8

(命令行)

```
# chown --help
Usage: chown [OPTION]... OWNER[.[GROUP]] FILE...
  or:  chown [OPTION]... .[GROUP] FILE...
Change the owner and/or group of each FILE to OWNER and/or GROUP.

  -c, --changes          be verbose whenever change occurs
  -h, --no-dereference affect symbolic links instead of any referenced file
                         (available only on systems with lchown system call)
  -f, --silent,--quiet suppress most error messages
  -R, --recursive        operate on files and directories recursively
  -v, --verbose          explain what is being done
      --help             display this help and exit
      --version          output version information and exit

Owner is unchanged if missing. Group is unchanged if missing, but changed
to login group
 if implied by a period. A colon may replace the period.

Report bugs to fileutils-bugs@gnu.ai.mit.edu
```



范例 1-9

(命令行)

```
1   $ ls -l filetest
    -rw-rw-r--   1 ellie   ellie        0 Jan 10 12:19 filetest
2   $ chown root filetest
    chown: filetest: Operation not permitted
3   $ su root
    Password:
4   # ls -l filetest
   -rw-rw-r--   1 ellie   ellie        0 Jan 10 12:19 filetest
5   # chown root filetest
6   # ls -l filetest
    -rw-rw-r--   1 root    ellie        0 Jan 10 12:19 filetest
7   # chown root:root filetest
8   # ls -l filetest
    -rw-rw-r--   1 root    root         0 Jan 10 12:19 filetest
```

说明

1. filetest 的用户和组为 ellie。
2. chown 命令只有超级用户(比如 root)才能使用。
3. 用户使用 su 命令将其身份改为 root。
4. 列表显示 filetest 的用户和组为 ellie。
5. 只有超级用户可以改变文件和目录的从属关系。filetest 的所有者被 chown 命令改为 root，但是组仍属于 ellie。
6. ls 命令的输出显示 root 现在是 filetest 的所有者。
7. 冒号(或者点号)用来表示用户 root 现在将组关系修改成 root。组名称在冒号后列出，中间没有空格。
8. filetest 的所有者和属组现在都为 root 了。

1.6.4 工作目录

登录成功之后，会在文件系统中得到一个工作目录，称为主目录(home directory)。工作目录会被 shell 派生的进程继承。shell 的任何一个子进程都可以改变自己的工作目录，但这个改变对父 shell 没有影响。

用于改变工作目录的命令 cd 是一个内置命令(参见范例 1-10)。每种 shell 都有自己的 cd 命令。内置命令是 shell 代码的一部分，由 shell 直接运行。运行内部命令时，shell 不调用 fork 和 exec。如果父 shell 创建了一个子 shell(脚本)，而 cd 命令是在子 shell 中执行的，则目录改变将在子 shell 中发生。子 shell 退出后，父 shell 将依然位于启动子 shell 之前所在的同一个目录中。

范例 1-10

```
1   > cd /
2   > pwd
    /
3   > bash
```

```
4   $ cd /home
5   $ pwd
    /home
6   $ exit
7   > pwd
    /
    >
```

说明

1. “>” 提示符是 TC shell 的提示符。cd 命令把目录改到/。cd 命令嵌在 shell 的内部代码中。
2. pwd 命令显示当前的工作目录为/。
3. 启动 bash shell。
4. cd 命令把目录改到/home。美元符号($)是 bash 的提示符。
5. pwd 命令显示当前的工作目录为/home。
6. 从 Bourne shell 中退出，返回到 TC shell。
7. 在 TC shell 中，当前目录依然是/。每个 shell 都有自己的 cd 命令。

1.6.5 变量

shell 可以定义两类变量：局部变量和环境变量。这些变量包含了用于定制 shell 的信息，以及其他进程正确运行所必需的信息。局部变量只属于创建它们的 shell，不会传给该 shell 派生的任何子进程。内置命令 set 可以显示 C shell 和 TC shell 的局部变量，以及 Bourne shell、bash 和 Kron shell 的两种变量。环境变量则正好相反，会被父进程传给子进程，子进程再传给孙进程，如此逐代传递。部分环境变量是登录 shell 从程序/bin/login 那继承而来，其他的则由用户的初始化文件、脚本或命令行创建。如果一个环境变量是在子进程设置的，它不会被回传给父进程。shell 命令 env 将显示环境变量，范例 1-11 列出了运行在 Solaris 5.9 上的 Bash shell 的环境变量。

范例 1-11

```
$ env
PWD=/home/ellie
TZ=US/Pacific
PAGER=less
HOSTNAME=artemis
LD_LIBRARY_PATH=/lib:/usr/lib:/usr/local/lib:/usr/dt/lib:/usr/openwin/lib
MANPATH=/usr/local/man:/usr/man:/usr/openwin/man:/opt/httpd/man
USER=ellie
MACHTYPE=sparc-sun-solaris2.9
TCL_LIBRARY=/usr/local/lib/tcl8.0
EDITOR=vi
LOGNAME=ellie
SHLVL=1
SHELL=/bin/bash
HOSTTYPE=sparc
OSTYPE=solaris2.9
```



```
HOME=/home/ellie
TERM=xterm
TK_LIBRARY=/usr/local/lib/tk8.0
PATH=/bin:/usr/bin:/usr/local/bin:/usr/local/etc:/usr/ccs/bin:/usr/etc:
/usr/ucb:/usr/local/X11:/usr/openwin/bin:/etc:.
SSH_TTY=/dev/pts/10
_=/bin/env
```

1.6.6 重定向与管道

文件描述符 所有I/O，包括文件、管道和套接字，都是由内核通过一种名为文件描述符的机制进行管理的。文件描述符是一个比较小的无符号整数，是文件描述符表的索引。文件描述符表由内核维护，内核用它访问打开的文件和I/O流。每个进程都会从它的父进程那继承文件描述符表。头3个文件描述符为0，1，2，被赋给了用户的终端。文件描述符0是标准输入(stdin)，1是标准输出(stdout)，2是标准错误输出(stderr)。当用户打开一个文件，而下一个可用的文件描述符是3时，这个文件描述符就被分配给新文件。如果所有文件符都被用掉了，就不能再打开新文件[8]。

重定向 当文件描述符被分配给终端以外的对象时，就被称为I/O重定向。shell把输出重定向到一个文件的过程是：先关闭标准文件描述符1(终端)，然后将这个描述符分配给该文件(见图1-5)。重定向标准输入时，shell先关闭文件描述符0(终端)，然后将它分配给一个文件(见图1-6)。Bourne、Bash和Korn shell重定向错误输出的方式是将一个文件分配给文件描述符2(见图1-7)。C/TC shell则不同，它要经过一个更复杂的过程(参见图1-8)。

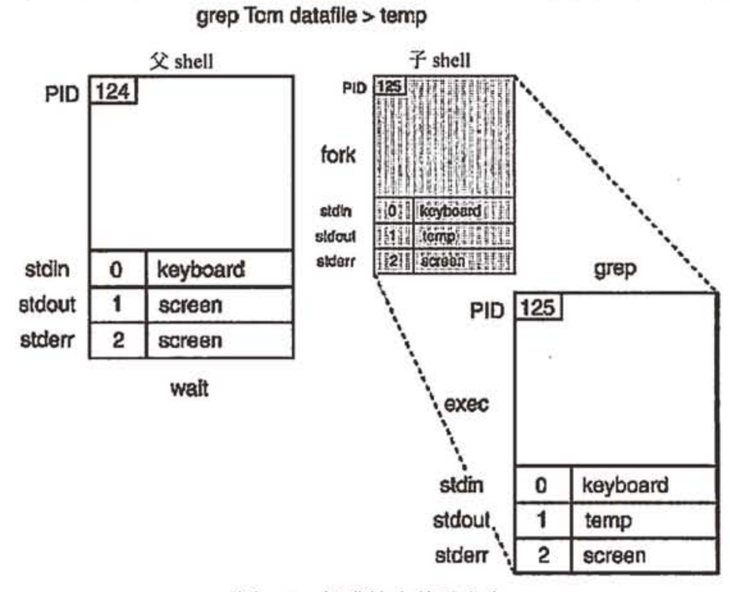


图 1-5 标准输出的重定向

[8] 参见内置命令 limit 和 ulimit。

mail tom < memo

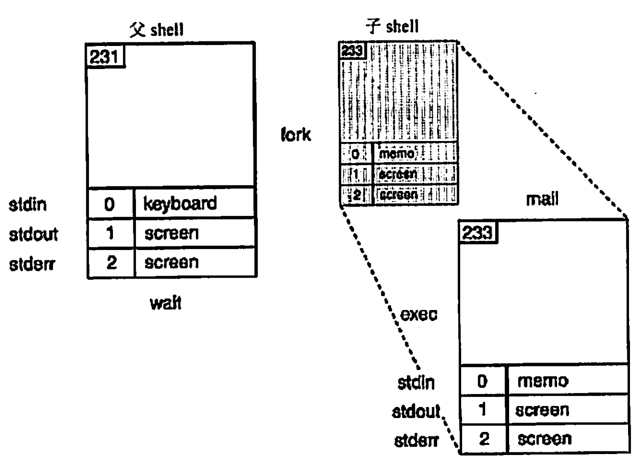


图 1-6 标准输入的重定向

find . -name filex 2> errors

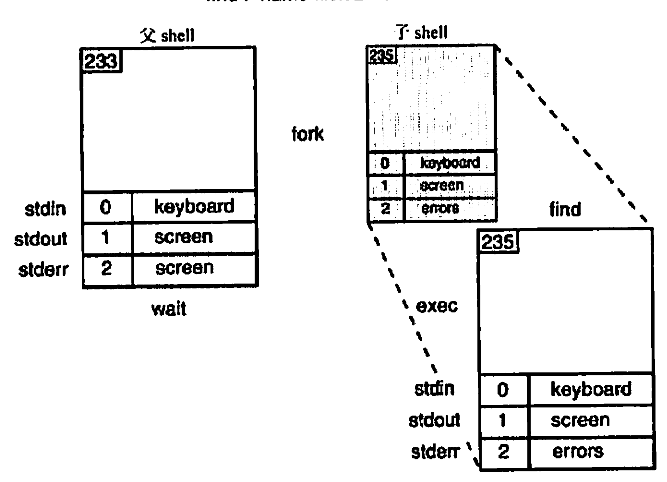


图 1-7 标准错误输出的重定向(用于 Bourne shell、Bash shell 和 Korn shell)

(find . -name filex > /dev/tty) >&errors

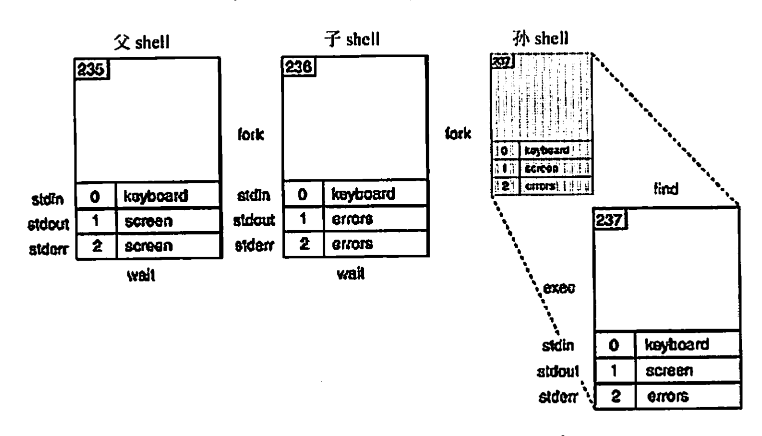


图 1-8 标准错误输出的重定向(用于 C shell 和 TC shell)



范例 1-12

```
1   $ who > file
2   $ cat file1 file2 >> file3
3   $ mail tom < file
4   $ find / -name file -print 2> errors
5   % ( find / -name file -print > /dev/tty) >& errors
```

说明

1. 命令 who 的输出从终端被重定向到 file(所有的 shell 都是以这种方式重定向输出)。

2. 命令 cat 的输出(连接 file1 和 file2)被附加到 file3 尾部(所有的 shell 都是以这种方式重定向和追加输出)。

3. file 的输入被重定向到 mail 程序，也就是说，file 的内容被送给用户 tom(所有的 shell 都是这样重定向输入)。

4. find 命令的所有错误都被重定向到文件 errors。输出则发往终端(Bourne shell 和 Korn shell 都以这种方式重定向错误输出)。

5. find 命令的所有错误都被重定向到文件 errors。输出则发往终端(C shell 和 TC shell 都这样重定向错误输出，%是 C shell 的提示符)。

管道 管道是 UNIX 进程间通信的最古老形式。管道用来将一条命令的输出传递给另一条命令作输入，这通常仅限于在父进程和子进程之间单向传递数据。shell 通过关闭和打开文件描述符来实现管道，但此时文件符并非分配给文件，而是赋给由系统调用 pipe 生成的管道描述符。创建管道文件描述符之后，父进程还要为管道中的每条命令派生一个子进程。子进程分别操纵管道描述符，一个进程往管道写数据，另一个则从管道读数据。

简单地说，管道其实只是一个内核缓冲区，两个进程通过它来共享数据，而不需要临时的中间文件。描述符设置好之后，管道中的两条命令同时被执行。其中一条命令的输出被发往缓冲区，当缓冲区已写满或该命令终止时，管道右边的命令开始读缓冲区。内核提供了这些动作的同步，因此，一个进程读/写管道时，另一个进程进入等待状态。

pipe 命令的语法为：

```
who | wc
```

为了在不使用管道的情况下完成同样的工作，需要 3 个步骤：

```
who > tempfile
wc tempfile
rm tempfile
```

而有了管道，shell 把 who 命令的输出作为输入发给 wc 命令。也就是说，管道左边的命令对管道进行写操作，而右边的命令对其进行读操作(参见图 1-9)。所有在命令行形式下将输出发送到屏幕的命令都可以作为写操作。ls、ps 和 date 命令就是这样的写操作。而从文件、键盘或管道读取输入的命令则视为读操作。sort、wc 和 cat 命令就是这样的读操作。

图 1-10 到 1-14 给出了管道的实现步骤。

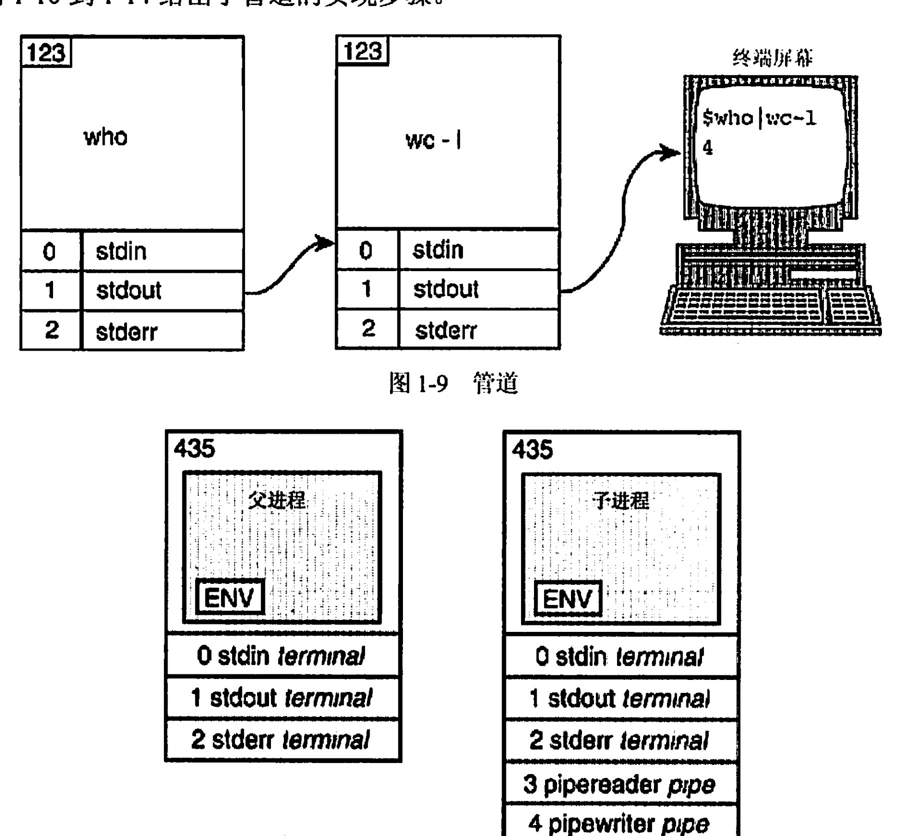


图 1-9 管道

图 1-10 父 shell 调用系统调用 pipe 来建立管道

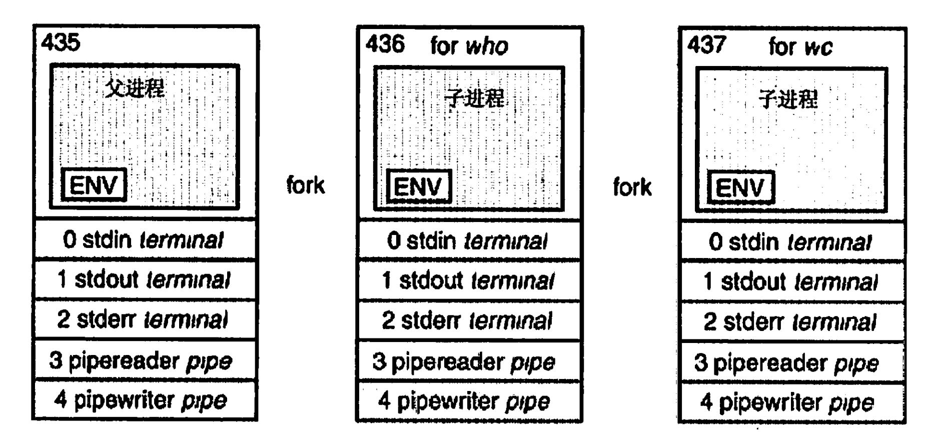


图 1-11 父进程派生两个子进程，每条命令对应一个



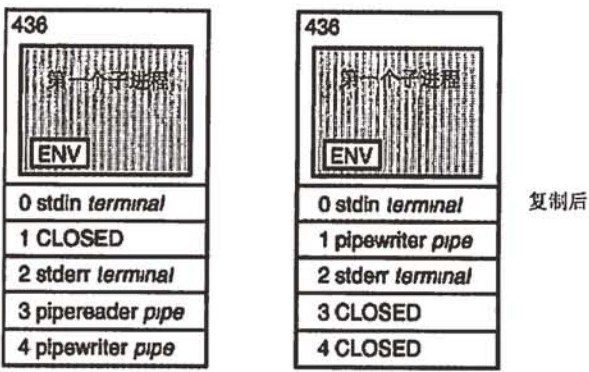


图 1-12　第一个子进程准备好了写管道

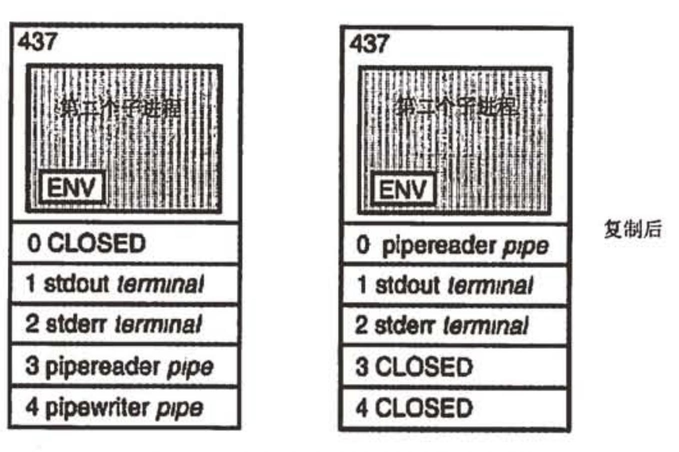


图 1-13　第二个子进程准备好了读管道

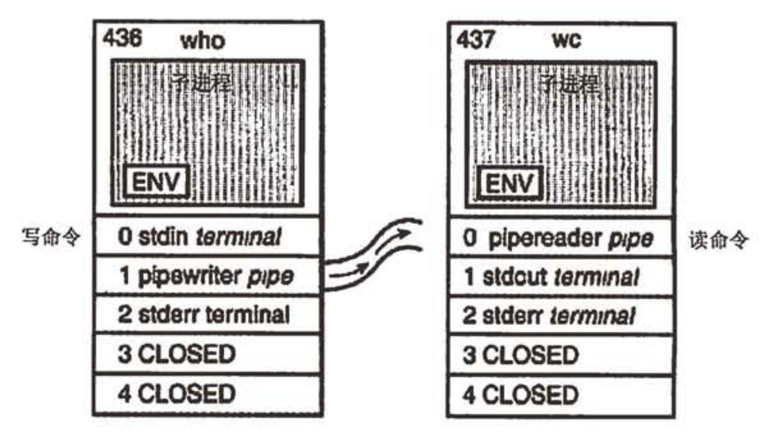


图 1-14　who 的输出被传递给 wc 的输入

1.6.7 shell 和信号

信号把消息发给进程，结果常常导致该进程终止，这通常是因为发生了段冲突、总线错误、电源故障或程序错误(比如，除数为 0 或无效的内存引用)这类意外事件。用户可以通过键入 Break、Delete、Quit 或 Stop 键来给进程发信号，而且所有共享终端的进程都会受所发出的信号的影响。还可以用 kill 命令终止进程。默认情况下，大部分信号会终止进程。每个进程对给定的信号可采取的动作如下：

(1) 忽略信号。

(2) 停止进程。

(3) 继续进程。

(4) 进程可以被程序中定义的函数所捕获。

Bourne、Bash 和 Korn shell 可以通过设置特定信号的相应动作，来处理进入程序中的信号(关于 Bourne shell 中信号处理，参见 8.8 节“捕获信号”。Korn shell 中的信号处理请参见 12.9 节“捕获信号”。Bash shell 中的信号处理请参见 14.8 节“捕获信号”)、或忽略信号。C 和 TC shell 只能处理^C(Ctrl+C 组合键)，即中断字符。

表 1-2 列出了进程可以使用的标准信号。

表 1-2 标准信号⑨

编　号	名　称	描　述	动　作
0	EXIT	shell 退出	终止
1	SIGHUP	终端已断开	终止
2	SIGINT	用户按了 Ctrl+C 组合键	终止
3	SIGQUIT	用户按了 Ctrl+\组合键	终止
4	SIGILL	非法硬件指令	程序错误
5	SIGTRAP	调试程序产生	程序错误
8	SIGFPE	算术错误，例如除数为 0	程序错误
9	SIGKILL	不能被捕获或忽略	终止
10	SIGUSR1	应用程序定义的用户信号	
11	SIGSEGV	无效的内存引用	程序错误
12	SIGUSR2	应用程序定义的用户信号	
13	SIGPIPE	断开管道连接	操作错误
14	SIGALRM	超时	发出警告
15	SIGTERM	程序终止	终止
17	SIGCHLD	子进程停止或死亡	忽略
18	SIGCONT	启动已停止的作业，此信号不能跳过	继续作业(已停止的)
19	SIGSTOP	停止一项作业，此信号不能跳过	停止进程

⑨ 可参考具体的系统手册。有关 UNIX 和 Linux 的信号的完整列表，可分别登录 www.cybermagician.co.uk/technet/unixsignals.htm 和 www.comptechdoc.org/os/linux/programming/Linux_pgsignals.html 网站进行查看。



(续表)

编 号	名 称	描 述	动 作
20	SIGSTP	交互停止，用户需按 Ctrl+Z 组合键	停止进程
21	SIGTTIN	后台作业尝试从控制终端读取信息	停止进程
22	SIGTTOU	后台作业尝试向控制终端写入信息	停止进程

1.7 在脚本中执行命令

shell 用作编程语言时，命令和 shell 程序可以通过编辑器输入并保存为文件，即脚本。文件的内容被 shell 逐行读取并执行。这些程序没有经过编译，需要通过解释来执行。编译过的程序在执行前被转换为机器语言。因此，shell 程序通常比二进制文件运行起来要慢一些，但是更容易编写，而且主要是用于简单任务的操作自动化。shell 程序的编写也可以在命令行上交互进行，如果任务很简单，这是最快捷的方式。但是，编写复杂脚本还是用编辑器方便些(除非你是打字高手)。下面这个脚本可以被任何一种 shell 执行，输出结果都一样。图 1-15 描述了一个名为 doit 的脚本的创建过程，并且说明它是如何与已有的 UNIX 程序、实用工具或命令结合起来的。

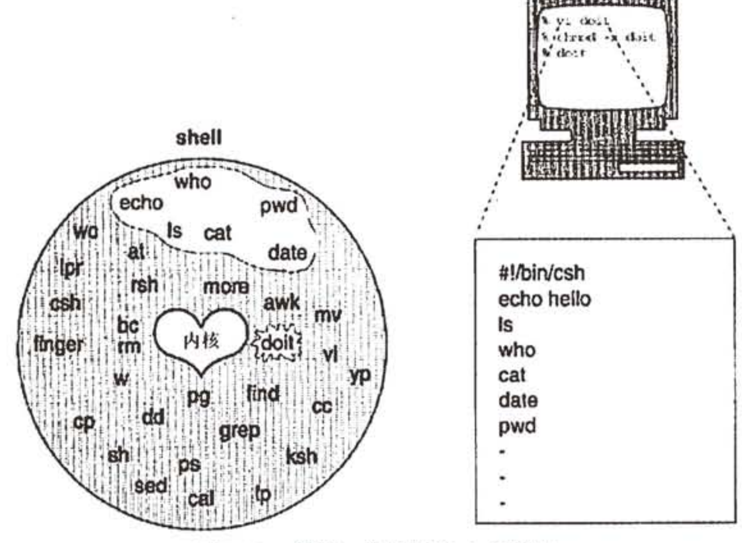


图 1-15 创建一个通用的 shell 脚本

说明

1. 挑选自己喜欢的编辑器，键入一组 UNIX/Linux 命令，每行输入一条。在第一行说明要用的是哪一种 shell，方法是在#！后输入所用 shell 的路径名。这个名为 doit 的程序将由 C shell 来执行。

2. 保存该文件，并且设置好它的执行权限，这样才能运行它。

3. 像执行其他 UNIX/Linux 命令一样执行这个程序。

chapter





shell 编程快速入门

2.1 shell 脚本简介

如果您需要读写或者维护程序，后面的实例将让您迅速了解到 shell 脚本结构和风格的概貌，并熟悉程序中出现的一些结构和语法。注意，如果对编程不熟悉，请跳过这一章直接阅读第 3 章。学会了如何编写脚本之后，再返回到这一章，可以对所学的编程知识进行一次快速的回顾。

C shell 和 TC shell 依照的是 C 语言的语法，相比之下，Bourne Shell 则基于老式的编程语言 Algol。

尽管 Bash shell 和 Korn shell 起源于 Bourne shell，但人们更喜欢将它们看作是 Bourne shell 和 C shell 的结合。

为了解释这些 shell 之间的差别，我们提供了 4 个实例程序，每个 shell(这里将 C shell 与 TC shell 看作一个)对应一个。每个程序都描述了所分析的 shell 的一些基本结构。

2.2 脚本实例：主要 shell 的比较

在每种特定 shell 的节末，有一个小程序用来说明如何编写一个完整的脚本。粗略看来，每个 shell 对应的程序都非常相似。不仅如此，并且它们都完成相同的工作，主要的区别就是语法。当您使用这些 shell 一段时间以后，您将很快适应这些差别并从中找出自己最喜欢的 shell。有关 C/TC、Bourne、Bash 和 Korn 这些 shell 之间的详细差别将在附录 B 中给出。

2.2.1 开始之前

必须很好地掌握 UNIX/Linux 命令。如果不知道基本的命令，将无法使用 shell 进行编



程。后续的三章将会教您如何使用 UNIX/Linux 的一些常用命令，本书最后面的附录 A 给出了最常用命令(也称作工具)的列表。

2.2.2 示例说明

每节末尾的示例脚本是向用户列表发送一封邮件，邀请他们去参加一个聚会。聚会的地点和时间由变量表示。来宾列表是从一个名为 guests 的文件中选择的。程序对这个文件是否存在进行检查，如果不存在，则程序退出。食品列表被存储在一个词表(数组)中。使用一个循环来遍历来宾列表，每个来宾将会收到一封电子邮件通知他们聚会的时间和地点，同时要求来宾带来食品列表中的一种食品。设置一个条件来判断是否存在名为 root 的来宾，如果他在来宾列表中，则被排除在外。也就是说，他将不会收到电子邮件邀请。循环一直进行直至来宾列表为空。循环每进行一次，就从食品列表中移走一项食品，这样每个来宾就会带来不同的食品。但是，如果来宾的数量比食品的种类要多的话，食品列表将被重置。这是由一个标准的循环控制语句处理的。

2.3 C shell 与 TC shell 的语法和结构

C shell 与 TC shell 的基本语法和结构在表 2-1 中列出。

表 2-1 C shell 与 TC shell 的语法和结构

shbang 行	shbang 行是脚本的第一行，它通知内核使用哪种 shell 解释脚本中的行。shbang 行由一个散列符号#，一个感叹号!(称作 bang)后接 shell 的完整路径组成，后面还可跟上各种 shell 选项。其他任何以#开头的行都被当作注释 例： `#!/bin/csh 或 #!/bin/tcsh`
注释	注释由一个符号# 后跟一些描述性的说明组成，它们可以从行的任意位置开始，到行的末尾结束，注释掉的语句程序将不予执行。 例： `# This is a comment`

通配符	shell 中有些字符的意义比较特殊，它们被称作是 shell 元字符或通配符。这些字符既非数字也非字母。例如，*、?和[ ]常用于文件名扩展；!是历史命令符，<、>、>>、<&和 | 符则用于标准 I/O 重定向和管道。为防止这些字符被 shell 解释，它们必须用反斜杠或引号进行引用。
	例：
	`rm *; ls ??; cat file[1-3]; !!`
	`echo "How are you?"`
	`echo Oh boy\!`

显示输出	echo 命令用于向屏幕显示输出。通配符必须使用反斜杠和配对引号进行转义。 例： `echo "Hello to you\!"`

(续表)

局部变量	局部变量的作用域被限定在当前 shell 中。当一个脚本执行结束或者 shell 退出后，它们不再可用。也就是说，它们超出了作用域。可以创建局部变量并为其赋值。
	例：
	`set variable_name = value`
	`set name = "Tom Jones"`

全局变量	全局变量又称环境变量。它们在当前运行的 shell 中创建，可由该 shell 派生的任意进程使用。一旦脚本结束或者定义该全局变量的 shell 退出，它将超出作用域。
	例：
	`setenv VARIABLE_NAME value`
	`setenv PRINTER Shakespeare`

提取变量值	使用美元符号可以从变量中提取数值。
	例：
	`echo $variable_name`
	`echo $name`
	`echo $PRINTER`

读取用户输入	特殊变量$<从用户输入中读取一行并将它赋给一个变量。
	例：
	`echo "What is your name?"`
	`set name = $<`

参数	可以从命令行传递参数给脚本。有两种方法可以在脚本中得到它们的值：位置参量和 argv 数组。
	例：
	`% scriptname arg1 arg2 arg3 ...`
	使用位置：
	`echo $1 $2 $3` arg1 赋给$1，arg2 赋给$2 等等
	`echo $*` 所有参数
	使用 argv 数组：
	`echo $argv[1] $argv[2] $argv[3]`
	`echo $argv[*]` 所有参数
	`echo $#argv` 参数个数

数组	数组是用空格隔开的一列词组成的词表，由一对圆括号括起来。内置的 shift 命令将词表左数第一个单词移开。与 C 语言不同的是，使用索引访问数组中的某个单词，索引值从 1 开始，而不是从 0 开始。
	例：
	`set word_list = ( word1 word2 word3 )`
	`set names = ( Tom Dick Harry Fred )`
	`shift names` 将 Tom 从词表中去掉
	`echo $word_list[1]` 显示词表中的第 1 个元素
	`echo $word_list[2]` 显示词表中的第 2 个元素
	`echo $word_list or $word_list[*]` 显示词表中的所有元素
	`echo $names[1]`
	`echo $names[2]`
	`echo $names[3]`
	`echo $names or echo $names[*]`



(续表)

<table>
<tr><td>命令替换</td><td>将 UNIX/Linux 命令的输出赋给一个变量，或者在字符串中使用某个命令的输出，命令由反引号引起来。
例：
set variable_name=`command`
echo $variable_name
set now = `date`　　反引号中的命令被执行，其输出赋给变量 now
echo $now
echo "Today is `date`"　　date 命令的输出被插入到字符串中</td></tr>
<tr><td>算术运算</td><td>保存算术运算结果的变量必须以一个@符号加一个空格开头。C shell 与 TC shell 只提供了整型算术运算。
例：
@ n = 5 + 5
echo $n</td></tr>
<tr><td>运算符</td><td>C shell 与 TC shell 中用于测试字符串和数字的运算符与 C 语言类似。
例：
等式运算符
==
!=
关系运算符
>　　大于
>=　　大于等于
<　　小于
<=　　小于等于
逻辑运算符
&&　　与
||　　或
!　　非</td></tr>
<tr><td>条件语句</td><td>if 结构后跟着一个用括号括起来的表达式。运算符类似于 C 运算符，关键字 then 放在闭括号之后。if 必须由 endif 结束。if/else if 等价于 switch 语句。
例：
if 结构
if (expression) then
　block of statements
endif

if/else 结构
if (expression) then
　block of statements
else
　block of statements
endif

if/else/else if 结构
if (expression) then
　block of statements
else if (expression) then
　block of statements
else if (expression) then</td></tr>
</table>

(续表)

条件语句(续)

```
      block of statements
   else
      block of statements
   endif
```

switch 结构

```
   switch variable_name
      case constant1:
         statements
      case constant2:
         statements
      case constant3:
         statements
   default:
         statements
   endsw

   switch ( "$color" )
      case blue:
         echo $color is blue
         breaksw
      case green:
         echo $color is green
         breaksw
      case red:
      case orange:
         echo $color is red or orange
         breaksw
      default:
         echo "Not a valid color"
   endsw
```

循环语句

有两种类型的循环，while 循环和 foreach 循环。

while 循环后跟着一个用圆括号括起来的表达式，一个语句段，最后以关键字 end 结束。只要表达式为真，循环将会一直持续。

foreach 循环后跟着一个变量，一个用圆括号括起来的词表，一个语句段，最后以关键字 end 结束。foreach 循环遍历词表，对每个词进行处理后将其移开，然后继续处理下一个词，当词表中所有词都被移走后，循环结束。

循环控制命令为 break 和 continue。

例：

```
   while ( expression )
      block of statements
   end

   foreach variable ( word list )
      block of statements
   end
   ------------------------------
   foreach color (red green blue)
      echo $color
   end
```



(续表)

文件测试	C shell 有一个内置的选项集可以用来测试文件属性，例如文件是否是目录文件、普遍文件(不是目录)、可读文件等。使用 UNIX test 命令还可以进行其他类型的文件测试。使用方法参见范例 2-1。 例： -r 当前用户可以读该文件 -w 当前用户可以写该文件 -x 当前用户可以执行该文件 -e 该文件存在 -o 该文件属于当前用户 -z 该文件长度为 0 -d 该文件是一个目录 -f 该文件是一个普通文件

范例 2-1

```
#!/bin/csh -f
1  if ( -e file ) then
       echo file exists
   endif

2  if ( -d file ) then
       echo file is a directory
   endif

3  if ( ! -z file ) then
       echo file is not of zero length
   endif

4  if ( -r file && -w file ) then
       echo file is readable and writable
   endif
```

C/TC shell 脚本

范例 2-2 中的程序是一个 C shell 和 TC shell 脚本的例子。这个程序包含了表 2-1 中提到的多种结构。

范例 2-2

```
1  #!/bin/csh -f
2  # The Party Program--Invitations to friends from the "guest" file
3  set guestfile = ~/shell/guests
4  if ( ! -e "$guestfile" ) then
       echo "$guestfile:t non-existent"
       exit 1
5  endif
6  setenv PLACE "Sarotini's"
7  @ Time = `date +%H` + 1
8  set food = ( cheese crackers shrimp drinks "hot dogs" sandwiches )
9  foreach person ( `cat $guestfile` )
```

```
10     if ( $person =~ root ) continue
11     mail -v -s "Party" $person << FINIS   # Start of here document
       Hi $person! Please join me at $PLACE for a party!
       Meet me at $Time o'clock.
       I'll bring the ice cream. Would you please bring $food[1] and
       anything else you would like to eat? Let me know if you can
       make it. Hope to see you soon.
            Your pal,
            ellie@`hostname`      # or 'uname -n'
12   FINIS
13      shift food
14      if ( $#food ==  0 ) then
           set food = ( cheese crackers shrimp drinks "hot dogs"
                        sandwiches )
        endif
15   end
     echo "Bye..."
```

说明

1. 这一行告诉内核正在运行的是一个 C shell 脚本。-f 是快速启动选项，它意味着“不要执行.cshrc 文件”。而通常情况下，每当一个新的 csh 程序启动时，会自动执行一个初始化文件。

2. 这是一条注释。shell 会忽略它，但对于想要了解脚本作用的人来说却是十分必要的。

3. 变量 guestfile 被赋值为 guests 文件的完整路径。

4. 这一行的含义是：如果文件 guests 不存在，那么它将向屏幕输出“guests non-existent”，然后从脚本中退出。退出状态值置为 1 以指出程序执行过程中出现了错误。

5. 这标志着 if 条件语句的结束。

6. 变量被赋值为地点和时间。PLACE 是一个环境变量。

7. Time 变量是一个局部变量。@符号通知 C shell 执行内置的算术运算。也就是说，在从 date 命令中提取出小时值后，将 Time 变量加 1。变量 Time 以大写的字母 T 开头以避免 shell 将这个变量与 shell 的保留字 time 混淆。

8. 创建 food 数组。它由一列以空格隔开的词组成，每个词均是 food 数组的一个元素。

9. foreach 循环由命令替换`cat $guestfile`创建的列表构成。cat 命令从一个文件中创建一个来宾列表。对来宾列表中除了用户名为 root 的每一个人，分别发出一封邮件邀请他们在给定的时间和地点参加聚会，同时要求他们带上食品列表上的一种食品。

10. 这个条件语句用于测试变量 person 的值是否与 root 相匹配。如果匹配，continue 语句将会使控制立即返回到循环的开头(而不是继续执行后续语句)。foreach 将处理词表中的下一个词。

11. 电子邮件消息的创建在 here 文档中完成。从用户定义的词 FINIS 到最后一个 FINIS 中的所有文本将发送给 mail 程序。foreach 循环遍历名字列表，执行从 foreach 到关键词 end 之间的所有指令。

12. FINIS 是一个用户定义的结束符，用于结束 here 文档。它包含了电子邮件消息的正文。



13. 每发送一个消息，shift 命令将食品列表左移，因此来宾列表中的下一位来宾将会带来食品列表中的下一种食品。

14. 如果食品列表为空，它将被重置，以确保其余的来宾均被指定带一种食品。

15. 这标志着循环语句的结束。

2.4 Bourne shell 的语法和结构

Bourne shell 的基本语法和结构如表 2-2 所示。

表 2-2 Bourne shell 的语法和结构

shbang 行	“shbang”行是脚本的第一行，它通知内核使用哪种 shell 解释脚本中的行。shbang 行由一个#!后跟 shell 的完整路径组成，后面还可跟上各种选项以控制 shell 的运行方式。 例： `#!/bin/sh`

注释	注释由一个符号# 后跟一些描述性的说明组成，它们可以从行的任意位置开始，在行的末尾结束。
	例：
	`# this text is not`
	`# interpreted by the shell`

通配符	shell 中有些字符的意义比较特殊，它们被称作是 shell 元字符或通配符。这些字符既非数字也非字母。例如，*、?和[ ]号常用于文件名扩展；<、>、2>、>>和 | 则用于标准 I/O 重定向和管道。为防止这些字符被 shell 解释，它们必须被引用。
	例：
	文件名扩展：
	`rm *; ls ??;  cat file[1-3];`
	引号保护元字符：
	`echo "How are you?"`

显示输出	echo 命令用于向屏幕显示输出。通配符必须使用反斜杠和配对引号进行转义。 例： `echo "What is your name?"`

局部变量	局部变量的作用域被限定在当前 shell 中。当一个脚本执行结束，它们不再可用。也就是说，它们超出了作用域。可以创建局部变量并为其赋值。
	例：
	`variable_name=value`
	`name="John Doe"`
	`x=5`

全局变量	全局变量又称环境变量。它们为当前运行的 shell 及由此 shell 派生的所有子进程创建。当脚本结束，该全局变量超出作用域。
	例：
	`VARIABLE_NAME=value`
	`export VARIABLE_NAME`
	`PATH=/bin:/usr/bin:.`
	`export PATH`

(续表)

提取变量值	使用美元符号可以从变量中提取数值。 例: echo $variable_name echo $name echo $PATH
读取用户输入	read 命令从用户输入中读取一行并将它赋给该命令右侧的一个或多个变量。read 命令可以接受多个变量名，每个变量被赋予一个单词。 例: echo "What is your name?" read name read name1 name2 ...
参数 (位置参数)	可以从命令行传递参数给脚本。位置参量用于在脚本中得到它们的值。 例: 命令行: $ scriptname arg1 arg2 arg3 ... 在脚本中: echo $1 $2 $3　　　　位置参量 echo $*　　　　所有的位置参量 echo $#　　　　位置参量的个数
数组 (位置参数)	Bourne shell 支持数组，使用位置参数也可以创建词表。set 内置命令后跟着一系列词，每个词都可以通过位置来访问。最多允许使用 9 个位置。内部命令 shift 将列表左侧第一个词从列表中移开。访问单个的词，位置的值从 1 开始。 例: set word1 word2 word3 echo $1 $2 $3　　　　显示 word1, word2 和 word3 set apples peaches plums shift　　　　移开 apples echo $1　　　　显示列表的第一个元素 echo $2　　　　显示列表的第二个元素 echo $*　　　　显示列表中的所有元素
命令替换	将 UNIX/Linux 命令的输出赋给一个变量，或者在字符串中使用某个命令的输出，命令由反引号引起来。 例: variable_name=`command` echo $variable_name now=`date` echo $now echo "Today is `date`"
算术运算	Bourne shell 不支持算术运算。必须使用 UNIX/Linux 命令进行计算。 例: n=`expr 5 + 5` echo $n



(续表)

<table>
<tr><td>运算符</td><td>Bourne shell 使用内置的 test 命令运算符来测试数字和字符串。
例：
等式运算符
=　字符串
!=　字符串
-eq　数字
-ne　数字

逻辑运算符
-a　与
-o　或
!　非

关系运算符
-gt　大于
-ge　大于等于
-lt　小于
-le　小于等于</td></tr>
<tr><td>条件语句</td><td>if 结构后跟着一个命令。如果要测试一个表达式，那么它必须用方括号括起来。关键字 then 放在闭括号之后。if 必须由 fi 结束。
例：
if 结构
if command
then
  block of statements
fi

if [expression]
then
  block of statements
fi

if/else 结构
if [expression]
then
  block of statements
else
  block of statements
fi

if/else/else if 结构
if command
then
  block of statements
elif command
then
  block of statements
elif command
then
  block of statements
else</td></tr>
</table>

(续表)

<table>
<tr><td>条件语句(续)</td><td><pre>
    block of statements
fi
----
if [expression]
then
    block of statements
elif [expression]
then
    block of statements
elif [expression]
then
    block of statements
else
    block of statements
fi
</pre>
case 命令结构
<pre>
case variable_name in
    pattern1)
        statements
            ;;
    pattern2)
        statements
            ;;
    pattern3)
            ;;
    *) default value
            ;;
esac
case "$color" in
    blue)
        echo $color is blue
        ;;
    green)
        echo $color is green
        ;;
    red|orange)
        echo $color is red or orange
        ;;
    *) echo "Not a color" # default
esac
</pre></td></tr>
<tr><td>循环语句</td><td>有 3 种类型的循环：while、until 和 for。
while 循环后跟一个命令或者一个用方括号括起来的表达式，关键字 do，一个语句段，最后以关键字 done 结束。只要表达式为真，do 和 done 之间的语句就会执行。
until 循环和 while 循环类似，不同的是，只要表达式为假，循环体就会执行。
for 循环用于在一列词中进行遍历，处理一个词然后将其移开，接着处理下一个词。当词表中所有的词都被处理后，循环结束。for 循环后跟一个变量名，接着是关键字 in，然后是一列词和一个语句段，最后以关键字 done 结束。
循环控制命令为 break 和 continue。</td></tr>
</table>



(续表)

<table>
<tr><td>循环语句(续)</td><td>例：<pre>
while command
do
  block of statements
done

while [expression]
do
  block of statements
done

 until command
do
   block of statements
done

until [expression]
do
  block of statements
done

for
variable in word1 word2 word3 ...
do
block of statements
done
</pre></td></tr>
<tr><td>文件测试</td><td>Bourne shell 使用 test 命令对条件表达式求值，它还有一个内置的选项集可用于测试文件属性，例如文件是否是目录文件、普通文件(不是目录)、可读文件等。参见范例 2-3。
例：<pre>
-d 该文件是一个目录
-f 该文件存在且不是一个目录
-r 当前用户可以读这个文件
-s 文件大小非 0
-w 当前用户可以写这个文件
-x 当前用户可以执行这个文件
</pre>范例 2-3<pre>
#!/bin/sh
1 if [-f file]
   then
       echo file exists
   fi

2 if [-d file]
   then
        echo file is a directory
   fi

3 if [-s file]
   then
       echo file is not of zero length
</pre></td></tr>
</table>

(续表)

<table>
<tr><td>文件测试(续)</td><td><pre>
     fi

   4 if [-r file -a -w file]
     then
          echo file is readable and writable
     fi
</pre></td></tr>
<tr><td>函数</td><td>函数允许您定义一段 shell 代码并对其命名。Bourne shell 引入了函数的概念，C shell 与 TC shell 则没有。
例：<pre>
function_name() {
   block of code
}
----

lister() {
   echo Your present working directory is `pwd`
   echo Your files are:
   ls
}
</pre></td></tr>
</table>

Bourne shell 脚本

范例 2-4

```
1   #!/bin/sh
2   # The Party Program--Invitations to friends from the "guest" file
3   guestfile=/home/jody/ellie/shell/guests
4   if [ ! -f "$guestfile" ]
5   then
6        echo "`basename $guestfile` non-existent"
7        exit 1
8   fi
9   PLACE="Sarotini's"; export PLACE
10  Time='date +%H'
    Time='expr $Time + 1'
11  set cheese crackers shrimp drinks "hot dogs" sandwiches
12  for person in `cat $guestfile`
    do
13       if [ $person =~ root ]
         then
14            continue
15       else
16       #    mail -v -s "Party" $person <<- FINIS
17            cat <<-FINIS
              Hi ${person}! Please join me at $PLACE for a party!
              Meet me at $Time o'clock.
              I'll bring the ice cream. Would you please bring $1 and
              anything else you would like to eat? Let me know if you
              can make it. Hope to see you soon.
                  Your pal,
                  ellie@'hostname'
```




```
            FINIS
18          shift
19          if [ $# -eq 0 ]
            then
20             set cheese crackers shrimp drinks "hot dogs" sandwiches
            fi
        fi
21   done
     echo "Bye..."
```

说明：

1. 这一行告诉内核正在运行的是一个Bourne shell脚本。

2. 这是一条注释。shell会忽略它，但对于想要了解脚本作用的人来说却是十分必要的。

3. 变量guestfile的值设为文件guests的完整路径。

4. 这一行的含义是：如果文件guests不存在，那么它将向屏幕输出"guests non-existent"然后从脚本中退出。

5. then通常另起一行，如果它的前面是个分号，则可以与if语句位于同一行。

6. UNIX basename命令将一个搜索路径中除文件名之外的部分删除。因为该命令处于反引号的内部，因此将执行命令替换，并用echo命令显示输出。

7. 如果文件不存在，程序将退出。退出状态值置为1以示程序有误。

8. 关键字fi标志着if语句块的结束。

9. 变量被赋值为地点和时间。PLACE 是一个环境变量，因为它刚被重新设置，所以输出。

10. Time变量的值是命令替换的结果。也就是说，命令date +%H(当前的钟点)的结果将赋给Time。

11. 用set命令将食品列表中要带的食品赋给一个特殊的变量(位置参数)。

12. 这是for循环。它进行遍历直至来宾文件中所列的每个人都被处理。

13. 如果变量person匹配用户名root，则循环控制转向for循环的开头并处理列表中的下一个人。用户root将不被邀请。

14. continue语句导致循环控制转向执行第12行，而不是继续执行第16行。

15. 如果第13行不为真则else中的语句段将会被执行。

16. 如果这行不被注释，邮件将被发送。最好将此行注释掉，直到整个程序都调试通过，否则每次测试脚本时都将向同一个人发送电子邮件。

17. 这条语句使用了cat命令和here文档来调试脚本，从而允许被测试的脚本向屏幕输出运行结果。正常情况下，如果第16行不被注释掉，这些输出内容是通过邮件发送的。

18. 每发送一个消息，shift命令就将食品列表左移，因此来宾列表上的下一位来宾将会带来食品列表中的下一种食品。如果食品列表为空，则被重置，以确保其余的来宾都能带一种食品。

19. $#的值是剩余位置参数的个数。如果个数为0，则食品列表为空。

20. 重置食品列表。

21. 关键字done标志着for循环语句的结束。

2.5 Korn shell 结构

Korn shell 和 Bash shell 十分相似，表 2-3 中的结构对两种 shell 都适用。要了解它们的细微差别，请参阅专门介绍各 shell 的章节。

表 2-3　Korn shell 的语法和结构

<table>
<tr><td>shbang 行</td><td>“shbang”行是脚本的第一行，它通知内核使用哪种 shell 解释脚本中的行。shbang 行由一个#!后跟 shell 的完整路径组成，后面还可跟上各种选项以控制 shell 的运行方式。
例：
#!/bin/ksh</td></tr>
<tr><td>注释</td><td>注释由一个符号# 后跟一些描述性的说明组成，它们可以从行的任意位置开始，到行的末尾结束。
例：
This program will test some files</td></tr>
<tr><td>通配符</td><td>shell 中有些字符的意义比较特殊，它们被称作是 shell 元字符或通配符。这些字符既非数字也非字母。例如，*、?和[ ]常用于文件名扩展，<、>、2>、>>和 | 则用于标准 I/O 重定向和管道。为防止这些字符被 shell 解释，它们必须被引用。
例：
rm *; ls ??; cat file[1-3];
echo "How are you?"</td></tr>
<tr><td>显示输出</td><td>echo 命令用于向屏幕显示输出。通配符必须使用反斜杠和配对引号进行转义。Korn shell 还提供了内置的 print 命令以替换 echo 命令。
例：
echo "Who are you?"
print "How are you?"</td></tr>
<tr><td>局部变量</td><td>局部变量的作用域被限定在当前 shell 中。当一个脚本执行结束或者 shell 退出后，它们不再可用。也就是说，它们超出了作用域。内置命令 typeset 也可以用来声明变量。可以创建局部变量并为其赋值。
例：
variable_name=value
typeset variable_name=value
name="John Doe"
x=5</td></tr>
<tr><td>全局变量</td><td>全局变量又称环境变量。它们为当前运行的 shell 及此 shell 派生的所有进程创建。当脚本结束或者定义该变量的 shell 退出后，该全局变量超出作用域。
例：
export VARIABLE_NAME =value
export PATH=/bin:/usr/bin:.</td></tr>
<tr><td>提取
变量值</td><td>使用美元符号可以从变量中提取数值。
例：
echo $variable_name
echo $name
echo $PATH</td></tr>
</table>



(续表)

<table>
<tr><td>读取
用户输入</td><td>用户可能要求输入，read 命令用于从用户输入中读取一行。如果 read 命令有多个参数，将会把这一行分成多个单词，每个单词赋给一个命名变量。Korn shell 允许将提示信息和 read 命令结合起来。
例：
<code>read name?"What is your name?"</code>　引号内部的是提示信息。当它显以后，read 命令等待用户输入。
<code>print -n "What is your name?"</code>
<code>read name</code>
<code>read name1 name2 ...</code></td></tr>
<tr><td>参数</td><td>可以从命令行传递参数给脚本。位置参数用于在脚本中得到它们的值。
例：
命令行中：
<code>$ scriptname arg1 arg2 arg3 ...</code>
在脚本里：
<code>echo $1 $2 $</code> 位置参数，$1 被赋值为 arg1，$2 被赋值为 arg2，...
<code>echo $*</code> 所有的位置参数
<code>echo $#</code> 位置参数的个数</td></tr>
<tr><td>数组</td><td>Bourne shell 利用位置参数创建词表。除了位置参数，Korn shell 还支持一种数组语法，因此其元素是通过从 0 开始的下标来访问的。Korn shell 数组使用 set –A 命令进行创建。
例：
<code>set apples pears peaches</code>　位置参数
<code>Print $1 $2 $3</code>　$1 为 apples，$2 为 pears　$3 为 peaches
<code>set -A array_name word1 word2 word3 ...</code>　数组
<code>set -A fruit apples pears plums</code>
<code>print ${fruit[0]}</code>　打印 apple
<code>${fruit[1]} = oranges</code>　赋一个新值</td></tr>
<tr><td>算术运算</td><td>Korn shell 支持整型运算。typeset -i 命令将声明一个新的整型变量。通过这种方式声明的变量可以进行整型运算。否则，算术运算将使用“(())”语法(即 let 命令)。
例：
<code>typeset -i variable_name</code>　声明整数
<code>typeset -i num</code>　num 被声明为一个整数
<code>num=5+4</code>
<code>print $num</code>　打印 9
<code>((n=5 + 5))</code>　let 命令
<code>print $n</code>　打印 10</td></tr>
<tr><td>命令替换</td><td>与 C shell、TC shell 以及 Bourne shell 类似，UNIX/Linux 命令的输出可以被赋给一个变量，或者通过使用反引号引用命令，在一个字符串中使用命令的输出。Korn shell 还提供了一种新的语法，无需反引号，只将命令包含在由美元符号开始的一对圆括号中即可。
例：
<code>variable_name=`command`</code>
<code>variable_name=$(command)</code>
<code>echo $variable_name</code>
<code>echo "Today is `date`"</code>
<code>echo "Today is $(date)"</code></td></tr>
</table>

(续表)

<table>
<tr>
<td>运算符</td>
<td>

与 C 语言运算符类似，Korn shell 使用内置的 test 命令运算符测试数字和字符串。

例：

等式运算符：		关系运算符：	
=	字符串，等于	>	大于
!=	字符串，不等于	>=	大于等于
==	数字，等于	<	小于
!=	数字，不等于	<=	小于等于

逻辑运算符：

&&	与
||	或
!	非

</td>
</tr>
<tr>
<td>条件语句</td>
<td>

if 结构后跟着一个由圆括号括起来的表达式。这个运算符与 C 运算符类似，关键字 then 放在闭括号之后。if 必须由 fi 结束。新的测试命令[[ ]]用于条件表达式的模式匹配。为和 Bourne shell 向后兼容，旧的测试命令[ ]仍旧可用。case 命令与 if/else 的功能等价。

例：

if 结构

```
if command
then
  block of statements
fi
----------------------------
if [[ string expression ]]
then
  block of statements
fi
----------------------------
if (( numeric expression ))
then
  block of statements
fi
```

if/else 结构

```
if command
then
  block of statements
else
  block of statements
fi
--------------------------
if [[ expression ]]
then
  block of statements
else
  block of statements
```

</td>
</tr>
</table>



(续表)

条件语句(续)	<pre>fi --------------------------- if ((numeric expression)) then block of statements else block of statements fi</pre> **if/else/else if 结构** <pre>if command then block of statements elif command then block of statements elif command then block of statements else block of statements fi --------------------------- if [[string expression]] then block of statements elif [[string expression]] then block of statements elif [[string expression]] then block of statements else block of statements fi ---------------------------- if ((numeric expression)) then block of statements elif ((numeric expression)) then block of statements elif ((numeric expression)) then block of statements else block of statements fi</pre> **case 结构** <pre>case variable name in</pre>

(续表)

<table>
<tr><td>条件语句
(续)</td><td>pattern1)
statements
;;
pattern2)
statements
;;
pattern3)
;;
esac
----
case "$color" in
blue)
echo $color is blue
;;
green)
echo $color is green
;;
red|orange)
echo $color is red or orange
;;
esac</td></tr>
<tr><td>循环语句</td><td>有 4 种类型的循环：while、until、for 与 select。
while 循环后跟一个用方括号括起来的表达式，关键字 do，一个语句段，最后以关键字 done 结束。只要表达式为真，do 和 done 之间的语句就会执行。
until 循环和 while 循环类似，不同的是，只要表达式为假，循环体就会执行。
for 循环用于在一列词中进行遍历，处理一个词然后将其移开，接着处理下一个词。当词表中所有的词都被处理后，循环结束。
select 循环常常提供一条提示信息(变量 PS3)和含多个选项的菜单，用户从中选择一项，这个输入将被存储在内置的特定变量 REPLY 中。select 循环通常与 case 命令一起使用。
循环控制命令为 break 和 continue。break 命令允许在到达末尾之前退出循环。continue 命令允许在到达末尾之前返回至循环表达式。
例：
while command
do
block of statements
done
----
while [[string expression]]
do
block of statements
done
----
while ((numeric expression))
do
block of statements
done

until command
do</td></tr>
</table>



(续表)

<table>
<tr><td>循环语句(续)</td><td><pre>
block of statements
done
----
until [[string expression]]
do
  block of statements
done
----
until ((numeric expression))
do
  block of statements
done

for variable in word_list
do
  block of statements
done
----
for name in Tom Dick Harry
do
  print "Hi $name"
done

select variable in word_list
do
  block of statements
done

PS3="Select an item from the menu"
for item in blue red green
  echo $item
done
</pre>
显示菜单：
1) 蓝色
2) 红色
3) 绿色</td></tr>
<tr><td>文件测试</td><td>Korn shell 使用 test 命令对条件表达式求值，它还有一个内置的选项集可以用来测试文件属性，例如文件是否是目录文件、普通文件(不是目录)、可读文件等。参见范例 2-5。
例：
-d　该文件是一个目录
-a　该文件存在且不是目录
–r　当前用户可以读这个文件
–s　文件大小非 0
–w　当前用户可以写这个文件
–x　当前用户可以执行这个文件</td></tr>
</table>

(续表)

<table>
<tr><td>文件测试
(续)</td><td>

范例 2-5

```
#!/bin/sh
1  if [ -a file ]
   then
     echo file exists
   fi

2  if [ -d file ]
   then
     echo file is a directory
   fi

3  if [ -s file ]
   then
     echo file is not of zero length
   fi

4  if [ -r file -a -w file ]
   then
      echo file is readable and writable
   fi
```

</td></tr>
<tr><td>函数</td><td>

函数允许您定义一段 shell 代码并对其命名。它有两种格式，一种来自 Bourne shell，一种是 Korn shell 版本，使用 function 关键字。

例：

```
function_name() {
   block of code
}

function function_name {
   block of code
}
-------------------------
function  lister {
   echo Your present working directory is `pwd`
   echo Your files are:
   ls
}
```

</td></tr>
</table>

Korn shell 脚本

范例 2-6

```
1   #!/bin/ksh
2   # The Party Program--Invitations to friends from the "guest" file
3   guestfile=~/shell/guests
4   if [[ ! -a "$guestfile" ]]
    then
        print "${guestfile##*/} non-existent"
```



```
        exit 1
    fi
5   export PLACE="Sarotini's"
6   (( Time=$(date +%H) + 1 ))
7   set -A foods cheese crackers shrimp drinks "hot dogs" sandwiches
8   typeset -i n=0
9   for person in $(< $guestfile)
    do
10      if  [[ $person = root ]]
        then
             continue
        else
             # Start of here document
11           mail -v -s "Party" $person <<- FINIS
             Hi ${person}! Please join me at $PLACE for a party!
             Meet me at $Time o'clock.
             I'll bring the ice cream. Would you please bring
             ${foods[$n]} and anything else you would like to eat? Let
             me know if you can make it.
                   Hope to see you soon.
                           Your pal,
                           ellie@`hostname`
             FINIS
12           n=n+1
13           if (( ${#foods[*]} == $n ))
             then
14             set -A foods cheese crackers shrimp drinks "hot dogs"
               sandwiches
             fi
        fi
15  done
    print "Bye..."
```

说明：

1. 这一行告诉内核正在运行的是一个 Korn shell 脚本。

2. 这是一条注释。shell 会忽略它，但对于想要了解脚本作用的人来说却十分必要。

3. 变量 guestfile 的值设为文件 guests 的完整路径。

4. 这一行的含义是：如果文件 guests 不存在，那么在屏幕上输出“guests non-existent”然后从脚本中退出。

5. 环境变量被赋值并输出(使得它对子 shell 可用)。

6. UNIX/Linux 命令的输出，即时间(小时数)赋值给变量 Time。变量被赋值为地点和时间。

7. 使用 set -A 命令将要带的食品列表赋给 foods 数组。这个列表上的每一项都可以用从 0 开始的索引来访问。

8. 用 typeset -i 命令来创建一个整数。

9. 对来宾列表上的每个来宾，发送一封邮件邀请他们在既定时间和地点参加聚会，并从食品列表中选出一项让他们带来。

10. 这个条件用于测试用户 root。如果用户是 root，则循环控制转向循环的开始并将来宾列表上的下一个来宾赋给 person 变量。

11. 邮件被发送。邮件的正文在 here 文档中。

12. 变量 n 加 1。

13. 如果数组中元素的个数等于变量的值，则已到达了数组的末尾。

14. 标志着循环语句的结束。

2.6 Bash shell 结构

Korn shell 与 Bash shell 十分类似，但仍有一些差别。bash 的结构在表 2-4 中列出。

表 2-4 Bash shell 的语法和结构

shbang 行	"shbang"行是脚本的第一行，它通知内核使用哪种 shell 解释脚本中的行。shbang 行由一个#!后跟 shell 的完整路径组成，后面还可跟上各种选项以控制 Shell 的运行方式。 例： `#!/bin/bash`
注释	注释由一个符号# 后跟一些描述性的说明组成，他们可以从行的任意位置开始，在行的末尾结束。 例： `# This is a comment`

通配符	shell 中有些字符的意义比较特殊，它们被称作是 shell 元字符或通配符。这些字符既非数字也非字母。例如，*、?和[ ]号常用于文件名扩展，<、>、2>、>>和 | 则用于标准 I/O 重定向和管道。为防止这些字符被 shell 解释，它们必须被引用。
	例：
	`rm *; ls ??; cat file[1-3];`
	`echo "How are you?"`

显示输出	echo 命令用于向屏幕显示输出。通配符必须使用反斜杠和配对引号进行转义。 例： `echo "How are you?"`

局部变量	局部变量的作用域被限定在当前 shell 中。当一个脚本执行结束，它们不再可用。也就是说，它们超出了作用域。内置函数 declare 也可以用来定义局部变量。可以创建局部变量并为其赋值。
	例：
	`variable_name=value`
	`declare variable_name=value`
	`name="John Doe"`
	`x=5`

全局变量	全局变量又称环境变量，它们由内置的 export 命令创建。它们为当前运行的 shell 及由此 shell 派生的所有子进程创建。当脚本结束，该全局变量超出作用域。
	带-x 选项的内置函数 declare 也用于设置环境变量并将其作为输出。
	例：
	`export VARIABLE_NAME=value`
	`declare -x VARIABLE_NAME=value`
	`export PATH=/bin:/usr/bin:.`



(续表)

提取
变量值
	使用美元符号可以从变量中提取数值。
	例：
	`echo $variable_name`
	`echo $name`
	`echo $PATH`

读取
用户输入
	用户可能会要求输入，read 命令用于从用户输入中读取一行。如果 read 命令有多个参数，则会将输入的这一行分解为多个单词，每个单词赋给一个命名变量。
	例：
	`echo "What is your name?"`
	`read name`
	`read name1 name2 ...`

参数	可以从命令行传递参数给脚本。位置参数用于在脚本中得到它们的值。 **例：** 命令行： `$ scriptname arg1 arg2 arg3 ...` 在脚本里： echo $1 $2 $3　　　位置参数 echo $*　　　　　所有位置参数 echo $#　　　　　位置参数的个数

数组	Bourne shell 利用位置参数创建词表。除了使用位置参数，Bash shell 还支持一种数组语法，因此其元素是通过从 0 开始的下标来访问的。Bash shell 数组使用 declare -a 命令进行创建。
	例：
	`set apples pears peaches  (positional parameters)`
	`echo $1 $2 $3`
	
	`declare -a array_name=(word1 word2 word3 ...)`
	`declare -a fruit=( apples pears plums )`
	`echo ${fruit[0]}`

命令替换	与 C shell、TC shell 以及 Bourne shell 类似，UNIX/Linux 命令的输出可以被赋给一个变量，或者通过使用反引号引用命令，在一个字符串中使用该命令的输出。Bash shell 还提供了一种新的语法：无需反引号，只将命令包含在由美元符号开始的一对圆括号中即可。
	例：
	``variable_name=`command` ``
	`variable_name=$( command )`
	`echo $variable_name`
	
	``echo "Today is `date`"``
	`echo "Today is $(date)"`

算术运算	Bash shell 支持整型运算。declare -i 命令将声明一个新的整型变量。为保持向后兼容，Korn shell 的 typeset 命令以后仍然可以使用。通过这种方式声明的变量可以进行整型运算。否则，算术运算将使用(())语法(let 命令)。 **例：**

(续表)

算术运算(续)

```
declare -i variable_name    用于 bash
typeset -i variable_name    用于保持与 ksh 的兼容

(( n=5 + 5 ))
echo $n
```

运算符

与 C 语言运算符类似，Bash shell 使用内置的 test 命令运算符测试数字和字符串。

例：

```
等式运算符          逻辑运算符
==    等于          &&     与
!=    不等于        ||     或
                    !      非
关系运算符
>            大于
>=           大于等于
<            小于
<=           小于等于
```

条件语句

if 结构后跟着一个由圆括号括起来的表达式。这个运算符与 C 运算符类似，关键字 then 放在闭括号之后。if 必须由 fi 结束。新的测试命令[[ ]]用于条件表达式的模式匹配。为和 Bourne shell 向后兼容，旧的测试命令[ ]仍旧可用。case 命令与 if/else 的功能等价。

例：

```
if 结构
if  command
then
   block of statements
fi

if  [[ expression  ]]
then
   block of statements
fi

if  (( numeric expression  ))
then
   block of statements
else
   block of statements
 fi

if/else 结构
if  command
then
   block of statements
else
   block of statements
fi

if  [[ expression ]]
then
   block of statements
```



(续表)

条件语句 (续)	<pre>else block of statements fi if ((numeric expression)) then block of statements else block of statements fi</pre>if/else/else if 结构<pre>if command then block of statements elif command then block of statements else if command then block of statements else block of statements fi ------------------------- if [[expression]] then block of statements elif [[expression]] then block of statements else if [[expression]] then block of statements else block of statements fi -------------------------- if ((numeric expression)) then block of statements elif ((numeric expression)) then block of statements else if ((numeric expression)) then block of statements else block of statements fi</pre>

(续表)

条件语句 (续)	case 结构 case variable_name in pattern1) statements ;; pattern2) statements ;; pattern3) ;; esac case "$color" in blue) echo $color is blue ;; green) echo $color is green ;; red|orange) echo $color is red or orange ;; *) echo "Not a matach" ;; esac
循环语句	有 4 种类型的循环：while、until、for 与 select。 while 循环后跟一个用方括号括起来的表达式，关键字 do，一个语句段，最后以关键字 done 结束。只要表达式为真，do 和 done 之间的语句就会执行。综合测试运算符[[ ]]是 Bash 的一个新运算符，为保持与 Bourne shell 的向后兼容，旧式的测试运算符[ ]仍然可以用来对条件表达式求值。 until 循环和 while 循环类似，不同的是，只要表达式为假，循环体就会执行。 for 循环用于在一列词中进行遍历，处理一个词然后将其移开，接着处理下一个词。当词表中所有的词都被处理后，循环结束。for 循环后跟一个变量名，关键字 in，一系列词，然后是一个语句段，最后以关键字 done 结束。 select 循环常常提供一条提示信息和含多个选项的菜单，用户从中选择项，这个输入将被存储在内置的特定变量 REPLY 中。select 循环通常与 case 命令一起使用。 循环控制命令为 break 和 continue。break 命令允许在到达末尾之前退出循环；continue 命令允许在到达末尾之前返回至循环表达式。 例： while command do block of statements done ------------------------- while [[string expression]] do block of statements done ------------------------- while ((numeric expression))



(续表)

循环语句(续)

```
  do
     block of statements
   done

until command
do
block of statements
done
--------------------------
until [[ string expression ]]
do
block of statements
done
--------------------------
until (( numeric expression ))
do
block of statements
done

for variable in word_list
do
  block of statements
done
--------------------------
for color in red green blue
do
    echo $color
done
--------------------------
select variable in word_list
do
    block of statements
done
 --------------------------
PS3="Select an item from the menu"
do  item in blue red green
  echo $item
done
```

显示菜单：

    蓝色
    红色
    绿色

函数

函数允许您定义一段 shell 代码并对其命名。它有两种格式，一种来自 Bourne shell，一种是 Bash shell 版本，使用 function 关键字。

例：

```
function_name() {
   block of code
}

function  function_name {
```

(续表)

函数(续)	`block of code`
	`}`
	`------------------------`
	`function lister {`
	`echo Your present working directory is `pwd``
	`echo Your files are:`
	`ls`
	`}`

Bash shell 脚本

范例 2-7

```
1   #!/bin/bash
    # GNU bash versions 2.x
2   # The Party Program--Invitations to friends from the "guest" file
3   guestfile=~/shell/guests
4   if [[ ! -e "$guestfile" ]]
    then
5       printf "${guestfile##*/} non-existent"
        exit 1
    fi
6   export PLACE="Sarotini's"
7   (( Time=$(date +%H) + 1 ))
8   declare -a foods=(cheese crackers shrimp drinks `"hot dogs"` sandwiches)
9   declare -i  n=0
10  for person in $(cat $guestfile)
    do
11      if  [[ $person == root ]]
        then
            continue
        else
            # Start of here document
12          mail -v -s "Party" $person <<- FINIS
            Hi $person! Please join me at $PLACE for a party!
            Meet me at $Time o'clock.
            I'll bring the ice cream. Would you please bring
            ${foods[$n] and anything else you would like to eat?
            Let me know if you can make it.
                  Hope to see you soon.
                       Your pal,
                       ellie@$(hostname)
            FINIS
13          n=n+1
14          if (( ${#foods[*]} ==  $n ))
            then
15             declare -a foods=(cheese crackers shrimp drinks `"hot dogs"`
                              sandwiches)
16          n=0
            fi
```




```
        fi
17  done
printf "Bye..."
```

说明：

1. 这一行告诉内核正在运行的是一个 Bash shell 脚本。

2. 这是一条注释。shell 会忽略它，但对于想要了解脚本作用的人来说却十分必要。

3. 变量 guestfile 赋值为文件 guests 的完整路径。

4. 这一行的含义是：如果文件 guests 不存在，那么它将向屏幕输出"guests non-existent"然后从脚本中退出。

5. 内置函数 printf 显示文件名(模式匹配)和字符串"non-existent"。

6. 环境(全局)变量被赋值并输出。

7. 一个数值表达式使用 UNIX/Linux date 命令的输出来得到当前时间。这个时间值被赋给变量 Time。

8. 定义(使用 declare -a)了一个含有若干元素的 Bash 数组 foods。

9. 定义了一个整数 n，初始值为 0。

10. 对来宾列表上除 root 用户外的每个人，发送一封邮件邀请他们在既定时间和地点参加聚会，并从食品列表中选出一项让他们带来。

11. 如果$person 的值为 root，则循环控制转向 for 循环的开始并从列表上的下一位来宾开始。

12. 发送邮件。邮件的正文在 here 文档中。

13. 整型变量 n 加 1。

14. 如果食品的数量等于数组的最大索引值，则列表已空。

15. foods 数组被重新赋值。每发送一个消息，将食品列表第一项移开以使得下一位来宾能带来食品列表中的下一种食品。如果人数比食品种类多，食品列表将被重置以保证每位来宾都能带来一种食品。

16. 数组索引的变量 n 被重置为 0。

17. 标志着循环语句的结束。





正则表达式与模式匹配

UNIX/Linux 系统上的工具数以百计，其中不乏像 ls、pwd、who 和 vi 这类常用的命令。正如木工们都有一套必备的工具一样，shell 编程人员也需要一套基本工具用于编写实用且高效的脚本。本书将详细讨论 3 种主要的工具：grep、sed 和 awk。这 3 种工具是 UNIX/Linux 上最重要的文本处理工具，用于处理来自管道输出或标准输入的文本。实际上，sed 和 awk 本身常常被用作脚本语言。要想完全认识 grep、sed 和 awk 的各种功能，必须先在正则表达式和正则表达式元字符的使用方面打好基础。本书的附录 A 中提供了一个完整的 UNIX/Linux 实用程序清单。

3.1 正则表达式

3.1.1 定义和示例

如果早已熟悉了正则表达式元字符的概念，就可以跳过这一节。但是，这些基础知识对于理解 grep、sed 和 awk 显示和处理数据的各种方法非常重要。

什么是正则表达式？正则表达式(regular expression，RE)[①]是一种字符模式，用于在查找过程中匹配指定的字符。在大多数程序里，正则表达式都被置于两个正斜杠之间；例如，“/love/”就是由正斜杠界定的正则表达式，它将匹配被查找的行中任何位置出现的相同模式。正则表达式的有趣之处在于，可以用特殊的元字符来控制它们。如果您是刚刚开始接触正则表达式，不妨看看下面这个例子，它能帮您理解整个概念。比如您用 vi 编辑器给您的朋友写邮件：

```
% vi letter
---------------------------------------------------------------------
Hi tom,
I think I failed my anatomy test yesterday. I had a terrible stomachache.
```

① 如果收到包含字符串 RE 的报错消息，就说明程序中使用的正则表达式出现了问题。




```
I ate too many fried green tomatoes.
Anyway, Tom, I need your help. I'd like to make the test up tomorrow, but
don't know where to begin studying. Do you think you could help me? After
work, about 7 PM, come to my place and I'll treat you to pizza in return
for your help. Thanks.
       Your pal,
       guy@phantom
~
~
~
~
--------------------------------------------------------------------------
```

现在，假如您发现 Tom 根本没有参加考试，但是 David 参加了。并且您注意到在问候部分，拼写 Tom 时使用了小写的 t。因此，您决定做一次全文替换，把邮件中出现的所有 tom 都换成 David，就像下面这样：

```
% vi letter
------------------------------------------------------------------------
Hi David,
I think I failed my anaDavidy test yesterday. I had a terrible sDavidachache.
I think I ate too many fried green Davidatoes.
Anyway, Tom, I need your help. I'd like to make the test up Davidorrow, but
don't know where to begin studying. Do you think you could help me? After
work, about 7 PM, come to my place and I'll treat you to pizza in return
for your help. Thanks.
       Your pal,
       guy@phanDavid
~
~
~
--> :1.$s/tom/David/g
------------------------------------------------------------------------
```

上例中，查找串使用的正则表达式是 tom；替换串中是 David。这条 vi 命令的意思是："从文件的第一行到文件末尾($)，用 David 替换所有行中出现的每一个 tom。"结果并非您所愿，anatomy、stomachache、tomatoes 和 tomorrow 中的"tom"被替换了，而 Tom 却没有被替换，这是因为您只要求用 David 替换 tom，却没有要求替换 Tom。怎么办？用正则表达式元字符吧。

3.1.2 正则表达式元字符

元字符是这样一类字符，它们表达的是不同于字面本身的含义。读者将在本书中学习两类元字符：shell 元字符和正则表达式元字符。它们各司其职。shell 元字符由 UNIX/Linux 的 shell 来解析。例如，当用户输入命令"rm *"时，命令中的星号就是一个 shell 元字符，称为通配符。shell 将其含义解析为"匹配当前工作目录下的所有文件名"。我们将在每种 shell 对应的相应章节中分别介绍它们使用的 shell 元字符。

正则表达式元字符则是由各种执行模式匹配操作的程序来解析，譬如 vi、grep、sed 和

awk[2]。正则表达式元字符是一类特殊的字符，可以通过它们以某种方式界定模式，从而控制进行哪些替换。可以使用元字符来定位在行首或行尾出现的单词，也可以用元字符指定任意字符或某一组字符，从而实现同时查找大写和小写字符，或只查找数字等操作。例如，如果要将名字 tom 和 Tom 都替换成 David，可以使用下面这条 vi 命令：

```
:1,$s/\<[Tt]om\>/David/g
```

这条命令的含义是："从文件的第一行到文件末尾(1,$)，用 David 替换单词 tom 或 Tom。"标志 g 表示全程执行该操作(即替换掉同一行中出现的所有指定模式)。正则表达式元字符\<和\>代表单词的开始和结束，[Tt]中这对方括号的含义是匹配括起来的任一字符(本例是匹配 T 或 t)。有 5 种基本的元字符可以被 UNIX\Linux 上所有的模式匹配工具识别。表 3-1 列出了可以在所有版本的 vi、ex、grep、egrep、sed 和 awk 中使用的正则表达式元字符。其他的元字符将在合适的时候结合相应的工具进行描述。

表 3-1 正则表达式元字符

元字符	功 能	示 例	匹 配 对 象
^	行首定位符	/^love/	匹配所有以 love 开头的行
$	行尾定位符	/love$/	匹配所有以 love 结尾的行
.	匹配单个字符	/l..e/	匹配包含一个 l，后跟两个字符，再跟一个 e 的行
*	匹配 0 或多个重复的位于星号前的字符	/ *love/	匹配包含跟在零个或多个空格后的模式 love 的行
[ ]	匹配一组字符中任一个	/[Ll]ove/	匹配包含 love 或 Love 的行
[x-y]	匹配指定范围内的一个字符	/[A-Z]ove/	匹配后面跟着 ove 的一个 A 至 Z 之间的字符
[^]	匹配不在指定组内的字符	/[^A-Z]/	匹配不在范围 A 至 Z 之间的任意一个字符
\	用来转义元字符	/love\./	匹配包含 love，后面跟一个句号。(未经转义的)句点通常匹配单个任意字符
许多使用 RE 元字符的 UNIX/Linux 程序都支持下面附加的元字符			
\<	词首定位符	/\<love/	匹配包含以 love 开头的词的行(vi 和 grep 支持)
\>	词尾定位符	/love\>/	匹配表含以 love 结尾的词的行(vi 和 grep 支持)
\(..\)	匹配稍后将要使用的字符的标签	/\(love\)able \1er/	最多可以使用 9 个标签，模式中最左边的标签是第一个。例如，模式 love 被保存为标签 1，用\1 表示。左边这个例子中，查找串是一个 lovable 后跟 lover 的长串(sed、vi 和 grep 支持)
x\{m\} 或 x\{m,\} 或 x\{m,n\}	字符 x 的重复出现：m 次、至少 m 次、至少 m 次且不超过 n 次[3]	o\{5, 10\}	匹配包含 5~10 个连续的字母 o 的行(vi 和 grep 支持)

② Korn shell 和 Bash shell 现在也提供了一组模式匹配元字符，类似于 grep、sed 和 awk 所使用的正则表达式元字符。

③ 并非所有版本的 UNIX 或所有的模式匹配工具都支持这组元字符，不过 vi 和 grep 通常都支持。



以下的介绍中，假定读者已经知道如何使用 vi 编辑器，我们通过 vi 查找字符串为例来描述每一个元字符。下面这些范例中，突出显示的字符用于指示 vi 查找的结果。

范例 3-1

（简单的正则表达式查找）

```
% vi picnic
---------------------------------------------------------------
I had a lovely time on our little picnic.
Lovers were all around us. It is springtime. Oh
love, how much I adore you. Do you know
the extent of my love? Oh, by the way, I think
I lost my gloves somewhere out in that field of
clover. Did you see them?  I can only hope love
is forever. I live for you. It's hard to get back in the
groove.
~
~
~
/love/
---------------------------------------------------------------
```

说明

上例使用的正则表达式是 love。查找到的模式 love 可能是自成一个单词，也可能是某个单词的一部分，例如 lovely、gloves 和 clover。

范例 3-2

（行首定位符(^)）

```
% vi picnic
---------------------------------------------------------------
I had a lovely time on our little picnic.
Lovers were all around us. It is springtime. Oh
love, how much I adore you. Do you know
the extent of my love? Oh, by the way, I think
I lost my gloves somewhere out in that field of
clover. Did you see them? I can only hope love
is forever. I live for you. It's hard to get back in the
groove.
~
~
~
/^love/
---------------------------------------------------------------
```

说明

脱字符(^)称为行首定位符。上例中，vi 只查找出那些在行首匹配到正则表达式 love 的行，即 love 必须是该行的头 4 个字符，它前面不能有任何字符，即便是空格。

范例 3-3

```
(行尾定位符($))
% vi picnic
------------------------------------------------------------------
I had a lovely time on our little picnic.
Lovers were all around us. It is springtime. Oh
love, how much I adore you. Do you know
the extent of my love? Oh, by the way, I think
I lost my gloves somewhere out in that field of
clover. Did you see them? I can only hope love
is forever. I live for you. It's hard to get back in the
groove.
~
~
~
/love$/
------------------------------------------------------------------
```

说明

美元符号($)称为行尾定位符。上例中，vi 只查找出那些在行尾匹配到正则表达式 love 的行，即 love 是该行的最后 4 个字符，后面紧跟一个换行符。

范例 3-4

```
(任意单个字符(.))
% vi picnic
------------------------------------------------------------------
I had a lovely time on our little picnic.
Lovers were all around us. It is springtime. Oh
love, how much I adore you. Do you know
the extent of my love? Oh, by the way, I think
I lost my gloves somewhere out in that field of
clover. Did you see them? I can only hope love
is forever. I live for you. It's hard to get back in the
groove.
~
~
~
/l.ve/
------------------------------------------------------------------
```

说明

句号(.)匹配除换行符外的任意单个字符。上例中，vi 查找包含由一个 l，后面跟一个任意字符，再跟一个 v 和一个 e 组成的那些正则表达式的行。查找结果是 love 和 live 的组合。

范例 3-5

```
(零个或多个前字符(*))
% vi picnic
------------------------------------------------------------------
I had a lovely time on our little picnic.
```



```
Lovers were all around us. It is springtime. Oh
love, how much I adore you. Do you know
the extent of my love? Oh, by the way, I think
I lost my gloves somewhere out in that field of
clover. Did you see them? I can only hope love
is forever. I live for you. It's hard to get back in the
groove.
~
~
~
/o*ve/
--------------------------------------------------------------
```

说明

星号(*)匹配零个或多个相同的前字符④。星号就好像是和它前面那个字符粘在一起的，它只控制这个字符。上面这个例子中，星号是和字母 o 粘在一起。它可以匹配单个字母 o 和任意多个连续字母 o，甚至还可以匹配根本不含字母 o 的模式。上例中，vi 查找在 0 个或多个连续字母 o 后紧跟一个 v 和一个 e 的组合，结果找到了 love、looove 和 lve 等。

范例 3-6

(一组字符([]))

```
% vi picnic
------------------------------------------------------------------------
I had a lovely time on our little picnic.
Lovers were all around us. It is springtime. Oh
love, how much I adore you. Do you know
the extent of my love? Oh, by the way, I think
I lost my gloves somewhere out in that field of
clover. Did you see them? I can only hope love
is forever. I live for you. It's hard to get back in the
groove.
~
~
~
/[Ll]ove/
------------------------------------------------------------------------
```

说明

方括号匹配某组字符中的一个。上例中，vi 查找包含一个大写或小写的 l 且后面跟着 ove 的正则表达式。

范例 3-7

(一个字符范围([-]))

```
% vi picnic
------------------------------------------------------------------------
```

④ 注意不要将这个元字符和 shell 的通配符(*)搞混。它们完全不同。shell 的星号匹配零个或多个任意字符，而正则表达式中的星号则匹配零个或多个相同的前字符。

```
I had a lovely time on our little picnic.
Lovers were all around us. It is springtime. Oh
love, how much I adore you. Do you know
the extent of my love? Oh, by the way, I think
I lost my gloves somewhere out in that field of
clover. Did you see them? I can only hope love
is forever. I live for you. It's hard to get back in the
groove.
~
~
~
/ove[a-z]/
---------------------------------------------------------------
```

说明

方括号里字符之间的连字符匹配某个范围内的一个字符。上例中，vi 查找包含 ove，后跟任何一个 ASCII 码的值在 a 和 z 之间的字符的正则表达式。由于指定的是 ASCII 码值的范围，所以不能写为[z-a]。

范例 3-8

(不在组内的字符([^]))

```
% vi picnic
------------------------------------------------------------------------
I had a lovely time on our little picnic.
Lovers were all around us. It is springtime. Oh
love, how much I adore you. Do you know
the extent of my love? Oh, by the way, I think
I lost my gloves somewhere out in that field of
clover. Did you see them? I can only hope love
is forever. I live for you. It's hard to get back in the
groove.
~
~
~
/ove[^a-zA-Z0-9]/
------------------------------------------------------------------------
```

说明

方括号内的脱字符是一个否定元字符。上例中，vi 查找的正则表达式包含 ove，后面紧跟一个特殊字符，这个字符既不在 ASCII 码范围 a 至 z 之间、又不在范围 A 至 Z 之间、也不是 0~9 之间的数字。例如，vi 找到的 ove 后面可能会跟着逗号、空格、句点等，因为这些字符都不在前面定义的那个集合内。

3.2 组合正则表达式元字符

基本的正则表达式元字符都已经介绍完了，现在可以把它们组合成更为复杂的表达式。



在下面例子中，括在正斜杠之间的正则表达式是查找串，用来跟文本文件中的每一行进行匹配。

范例 3-9

注意，行号是为了方便叙述而加上的，并不是文本文件内容的一部分，竖线则是用来标识左、右页边。

```
   --------------------------------------------------------------
1 |Christian Scott lives here and will put on a Christmas party.|
2 |There are around 30 to 35 people invited.                    |
3 |They are:                                                    |
4 |                                                          Tom|
5 |Dan                                                          |
6 |  Rhonda Savage                                              |
7 |Nicky and Kimberly.                                          |
8 |Steve, Suzanne, Ginger and Larry.                            |
   --------------------------------------------------------------
```

说明

a. /^[A-Z]..$/ 查找文本中所有以大写字母开头、后跟两个任意字符，再跟一个换行符的行。查找结果是第 5 行的 Dan。

b. /^[A-Z][a-z]*3[0-5]/ 查找所有以大写字母开头、后跟零个或多个小写字母或空格，再跟数字 3 和一个 0~5 之间的数字的行。查找结果是第 2 行。

c. /[a-z]*\./ 查找包含跟在零个或多个小写字母后的句点的行。结果是第 1、2、7、8 行。

d. /^ *[A-Z][a-z][a-z]$/ 查找以零个或多个空格开头(注意：制表符不算空格)，后跟一个大写字母、两个小写字母和一个换行符的行。结果将是第 4 行的 Tom 和第 5 行的 Dan。

e. /^[A-Za-z]*[^,][A-Za-z]*$/ 查找以零个或多个大/小写字母开头，后跟一个非逗号的字符，再跟零个或多个大/小写字母和一个换行符的行。结果是第 5 行。

其他正则表达式元字符

下面将要介绍的这些元字符不一定适用于所有使用正则表达式的工具，但可用于 vi 编辑器和某些版本的 sed 和 grep。此外还有一种被 egrep 和 awk 使用的扩展元字符集，我们将在后续章节中讨论。

范例 3-10

(词首定位符 (\<) 和词尾定位符 (\>))

```
% vi textfile
    --------------------------------------------------------
    Unusual occurrences happened at the fair.
--> Patty won fourth place in the 50 yard dash square and fair.
    Occurrences like this are rare.
    The winning ticket is 55222.
    The ticket I got is 54333 and Dee got 55544.
    Guy fell down while running around the south bend in his last event.
    ~
    ~
```

```
~
/\<fourth\>/
-------------------------------------------------------------
```

说明

这个例子将找出每一行中的单词 fourth。\<是词首定位符，\>则是词尾定位符。单词的存在形式可以是：以空格分隔、由标点终结、开始于行首、结束于行尾等。

范例 3-11

```
% vi textfile
   -------------------------------------------------------------
   Unusual occurrences happened at the fair.
--> Patty won fourth place in the 50 yard dash square and fair.
   Occurrences like this are rare.
   The winning ticket is 55222.
   The ticket I got is 54333 and Dee got 55544.
--> Guy fell down while running around the south bend in his last event.
   ~
   ~
   ~
   /\<f.*th\>/
   -------------------------------------------------------------
```

说明

查找以 f 开头、后跟 0 个或多个任意字符(.*)，再跟以 th 结尾的字符串的任意单词(或词组)。

范例 3-12

(用\(和\)记录模式)

```
% vi textfile (Before substitution)
   -------------------------------------------------------------
   Unusual occurences happened at the fair.
   Patty won fourth place in the 50 yard dash square and fair.
   Occurences like this are rare.
   The winning ticket is 55222.
   The ticket I got is 54333 and Dee got 55544.
   Guy fell down while running around the south bend in his last event.
   ~
   ~
   ~
1  :1,$s/\([Oo]ccur\)ence/\1rence/
   -------------------------------------------------------------

   % vi textfile (After substitution)
   -------------------------------------------------
--> Unusual occurrences happened at the fair.
   Patty won fourth place in the 50 yard dash square and fair.
--> Occurrences like this are rare.
   The winning ticket is 55222.
```



```
    The ticket I got is 54333 and Dee got 55544.
    Guy fell down while running around the south bend in his last event.
    ~
    ~
    ~
    ------------------------------------------------------------------
```

说明

编辑器查找完整的字符串 occurence 或 Occurence(注：此处故意将其拼错)。如果找到了，就把圆括号中的这部分模式加上标签(即将 occur 或 Occur 标记)。这是第一个被标记的模式，因此被称为标签 1。这个模式被保存在标记为 1 的内存寄存器中。执行替换时，先将\1 替换为寄存器的内容，然后加上单词的剩余部分 rence。这样，开始时的 occurence 最后就被替换为 occurrence，参见图 3-1。

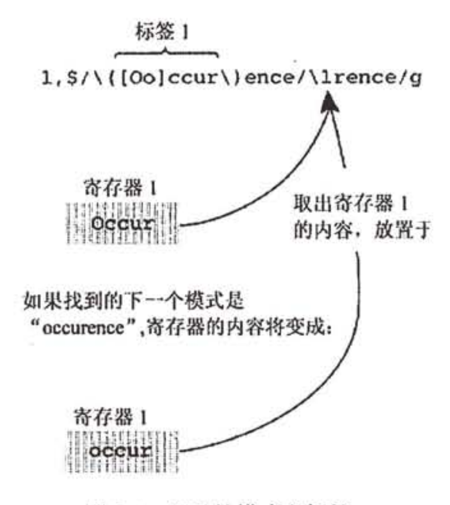


图 3-1 记录的模式和标签

范例 3-13

```
% vi textfile (Before substitution)
  ------------------------------------------------------------------
    Unusual occurrences happened at the fair.
    Patty won fourth place in the 50 yard dash square and fair.
    Occurrences like this are rare.
    The winning ticket is 55222.
    The ticket I got is 54333 and Dee got 55544.
    Guy fell down while running around the south bend in his last event.
    ~
    ~
    ~
1   :s/\(square\) and \(fair\)/\2 and \1
    ------------------------------------------------------------------

    % vi textfile (After substitution)
    ------------------------------------------------------------------
   Unusual occurrences happened at the fair.
-->  Patty won fourth place in the 50 yard dash fair and square
```

```
    Occurrences like this are rare.
    The winning ticket is 55222.
    The ticket I got is 54333 and Dee got 55544.
    Guy fell down while running around the south bend in his last event.
    ~
    ~
    ~
    ----------------------------------------------------------------
```

说明

编辑器查找正则表达式 square and fair，将其中的 square 标记为标签 1，fair 标记为标签 2。执行替换时，\1 被寄存器 1 的内容替换，\2 被寄存器 2 的内容替换。参见图 3-2。

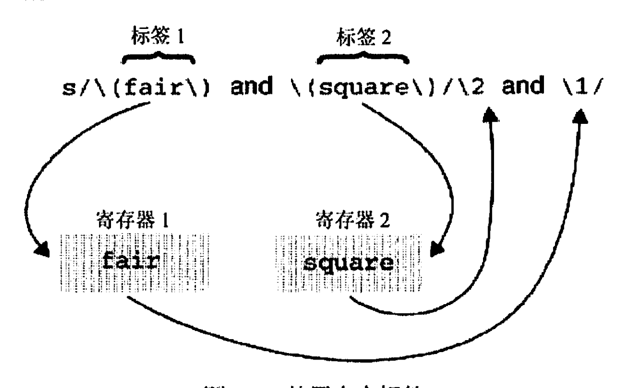


图 3-2 使用多个标签

范例 3-14

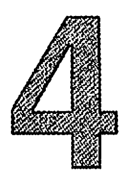
```
(模式的重复(\{n\}))
% vi textfile
    ---------------------------------------------
    Unusual occurrences happened at the fair.
    Patty won fourth place in the 50 yard dash square and fair.
    Occurrences like this are rare.
--> The winning ticket is 55222.
    The ticket I got is 54333 and Dee got 55544.
    Guy fell down while running around the south bend in his last
    event.
    ~
    ~
    ~
    ~
1   /5\{2\}2\{3\}\./
------------------------------------------------------
```

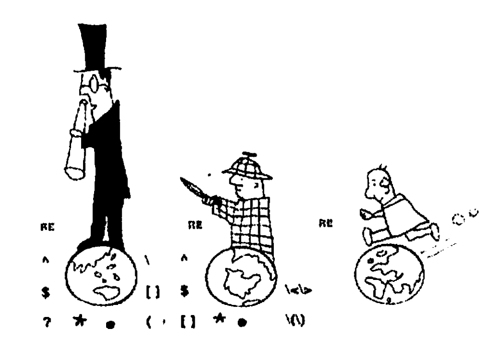
说明

1. 查找包含两个数字 5，后跟 3 个数字 2，再跟一个句点的行。



chapter 4





grep 家族

grep 家族由命令 grep、egrep 和 fgrep 组成。grep 命令在文件中全局查找指定的正则表达式，并且打印所有包含该表达式的行。egrep 和 fgrep 都只是 grep 的变体。egrep 命令是扩展的 grep，支持更多的正则表达式元字符。fgrep 命令被称为固定 grep(fixed grep)，有时也被称作快速 grep(fast grep)，它按字面解释所有的字符，也就是说，正则表达式元字符不会被特殊处理，它们只匹配自己。自由软件基金会提供了 grep 的免费版本，称作 GNU grep。Linux 系统上使用的就是这种版本的 grep。在 Sun 公司的 Solaris 操作系统上也可以找到 GNU 版的 grep，所在目录是/usr/xpg4/bin。GNU 版的 grep 扩展了基本的正则表达式元字符集，增加了与 POSIX 的一致性，并且包括很多新的命令行选项。它们还提供了一个名为 rgrep 的递归式 grep，用于逐级搜索整个目录树。

4.1 grep 命令

4.1.1 grep 的含义

grep 这个名字的由来可以追溯到 ex 编辑器。如果启动 ex 编辑器来查找某个字符串，需要在 ex 的命令提示符后键入：

```
:/pattern/p
```

字符“p”表示打印(print)命令，所以包含字符串 pattern 的第一行的内容会被打印出来。如果希望打印所有包含 pattern 的行，可以输入：

```
:/g/pattern/p
```

当 g 出现在 pattern 前面时，其含义是“文件中的所有行”，或“执行全局替换”。

被查找的模式称作正则表达式(regular expression)，因此我们可以用 RE 来替换 pattern，于是上面这条命令就变成了：

```
:g/RE/p
```



说到这，您就应该清楚 grep 的含义和它名称的来源了。它表示“全局查找正则表达式(RE)并且打印结果行。”使用 grep 的好处在于，不需要启动编辑器就可以执行查找操作，也用不着把正则表达式括在正斜杠里。使用 grep 比使用 ex 或 vi 快得多。

4.1.2 grep 如何工作

grep 命令在一个或多个文件中查找某个字符模式。如果这个模式中包含空格，就必须用引号把它括起来。grep 命令中，模式可以是一个被引号括起来的字符串，也可以是单个词[①]，位于模式之后的所有单词都被视为文件名。grep 将输出发送到屏幕，它不会对输入文件进行任何修改或变化。

命令格式

```
grep word filename filename
```

范例 4-1

```
grep Tom /etc/passwd
```

说明

grep 将在文件/etc/passwd 中查找模式 Tom。如果查找成功，文件中的相应行会显示在屏幕上，如果没有找到指定的模式，就不会有任何输出，如果所指定的文件不是一个合法文件，屏幕上就会显示报错信息。如果发现了要查找的模式，grep 返回的退出状态为 0，表示成功，如果没找到，返回的退出状态就是 1，而找不到指定的文件时，退出状态将是 2。

grep 程序的输入可以来自标准输入或管道，而不仅仅是文件。如果忘了指定文件，grep 会以为你要它从标准输入(即键盘)获取输入，于是停下来等着你键入一些字符。如果输入来自管道，就会有另一条命令的输出通过管道变成 grep 命令的输入，如果匹配到要查找的模式，grep 会把输出打印在屏幕上。

范例 4-2

```
ps -ef | grep root
```

说明

ps 命令的输出(ps –ef 显示正在系统上运行的所有进程)被送到 grep，然后所有包含 root 的行被打印在屏幕上。

4.1.3 元字符

元字符也是一种字符，但它表达的含义不同于字符本身的字面含义。例如，^和$就是元字符。

grep 命令支持很多正则表达式元字符(参见表 4-1)，以使用户更精确地定义要查找的模式。它还提供了很多命令选项(参见表 4-2)用于调整执行查找或显示结果的方式。例如，可以通过指定选项来关闭大小写敏感、要求显示行号，或者只显示报错信息等。

---

① 词(word)也称为标记(token)。

范例 4-3

```
grep -n '^jack:' /etc/passwd
```

说明

grep 在文件/etc/passwd 中查找 jack，如果 jack 出现在某行的行首，grep 就打印出该行的行号和内容。

表 4-1 grep 使用的正则表达式元字符

元字符	功　能	示　例	匹 配 对 象
^	行首定位符	'^love'	匹配所有 love 开头的行
$	行尾定位符	'love$'	匹配所有以 love 结尾的行
.	匹配一个字符	'l..e'	匹配包含一个 l，后跟两个字符，再跟一个 e 的行
*	匹配零个或多个前导字符	' *love'	匹配包含跟在 0 或多个空格后的模式 love 的行
[ ]	匹配一组字符中任一个	'[Ll]ove'	匹配 love 或包含 Love 的行
[^]	匹配不在指定字符组内的字符	' [^A-K]ove'	匹配包含一个不在 A 至 K 之间的字符，并且该字符后紧跟着 ove 的行
\<	词首定位符	'\<love'	匹配包含以 love 开头的词的行
\>	词尾定位符	'love\>'	匹配表含以 love 结尾的词的行
\(..\)	标记匹配到的字符	'\(love\)ing'	标记寄存器中的一段字符，被记作 1 号寄存器。如果之后要引用这段字符，可以用\1 来重复该模式。最多可以设置 9 个标签，从左边开始编号，最左边的是第一号。左边这个例子中，模式 love 被保存在 1 号寄存器里，之后可以用\1 来引用它
x\{m\} 或 x\{m,\} 或 x\{m,n\}[②]	字符 x 的重复出现次数：m 次、至少 m 次、至少 m 次但不超过 n 次	'o\{5\}' 'o\{5,\} ' 'o\{5, 10\}'	匹配连续出现 5 个 o、至少 5 个 o 或 5~10 个 o 的行

表 4-2 grep 的选项

选　项	功　能
-b	在每一行前面加上其所在的块号，根据上下文定位磁盘块时可能会用到
-c	显示匹配到的行的数目，而不显示行的内容
-h	不显示文件名
-i	比较字符时忽略大小写的区别(即认为字母的大小写相等)
-l	只列出匹配行所在文件的文件名(每个文件名只列一次)，文件名之间用换行符分隔

② 并非所有版本的 UNIX 或所有的模式匹配工具都支持元字符\{ \}，但 vi 和 grep 通常都支持它们。



(续表)

选　项	功　能
-n	在每一行前面加上它在文件中的相对行号
-s	无声操作，即只显示报错信息，以检查退出状态
-v	反向查找，只显示不匹配的行
-w	把表达式作为词来查找，就好像它被\<和\>所包含一样。只适用于 grep(并非所有版本的 grep 都支持这一功能，譬如，SCO UNIX 就不支持)

4.1.4 grep 的退出状态

grep 在 shell 脚本中很有用，因为它总会返回一个退出状态，以说明能否定位到要查找的模式或文件。如果找到了模式，grep 返回的退出状态为 0，表示成功，如果找不到模式，grep 返回 1 作为退出状态，而当找不到要搜索的文件时，grep 返回的退出状态是 2(其他查找模式的 UNIX 工具，例如 sed 和 awk，不使用退出状态来说明查找模式成功与否，它们只在命令中出现语法错误时才报告失败)。

下面这个范例中，在文件/etc/passwd 中未找到 john。

范例 4-4

```
1   % grep 'john' /etc/passwd        # john is not in the passwd file
2   % echo $status       (csh)
      1
```

或者

```
2   $ echo $?           (sh, ksh)
1
```

说明

1. grep 在文件/etc/passwd 中查找 john，如果成功，grep 以状态 0 退出。若未在该文件中找到 john，grep 以状态 1 退出。如果没找到文件/etc/passwd，返回的退出状态是 2。

2. C shell 的变量 status 和 Bourne/Korn shell 的变量“？”的值都是上一条命令执行后的退出状态。

4.2 使用正则表达式的 grep 实例

下面这些例子所使用的文件的名字是 datafile。为了方便您的阅读，将会在必要时重复地出现。

% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 4-5

```
% grep NW datafile
northwest          NW      Charles Main          3.0   .98   3      34
```

说明

打印文件 datafile 中所有包含正则表达式 NW 的行。

范例 4-6

```
% grep NW d*
datafile: northwest  NW     Charles Main       3.0   .98    3    34
db: northwest        NW     Joel Craig         30    40     5    123
```

说明

打印所有名字以 d 开头的文件中，包含正则表达式 NW 的所有行。shell 把 d*扩展为所有名字以 d 开头的文件，这个例子中，所包括的文件是 db 和 datafile。

范例 4-7

```
% grep '^n' datafile
northwest          NW      Charles Main       3.0   .98    3    34
northeast          NE      AM Main Jr.        5.1   .94    3    13
north              NO      Margot Weber       4.5   .89    5     9
```

说明

打印所有以字母 n 开头的行。脱字符(^)是句首定位符。

范例 4-8

```
% grep '4$' datafile
northwest          NW      Charles Main          3.0   .98    3     34
```

说明

打印所有以数字 4 结尾的行。美元符($)是行尾定位符。

范例 4-9

```
% grep TB Savage datafile
```



```
grep: Savage: No such file or directory
datafile: eastern     EA      TB Savage              4.4   .84    5   20
```

说明

第一个参数是模式，其余所有参数都是文件名，grep 将在 Savage 和 datafile 这两个文件中查找 TB。如果要查找 TB Savage，请看下一个范例。

% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 4-10

```
% grep 'TB Savage' datafile
eastern           EA       TB Savage             4.4   .84    5   20
```

说明

打印所有包含模式 TB Savage 的行。如果不用引号(这个例子中，使用单引号或双引号都可以)，TB 和 Savage 之间的空格将导致 grep 在文件 Savage 和 datafile 中查找 TB，就像前面那个示例一样。

范例 4-11

```
% grep '5\..' datafile
western           WE      Sharon Gray        5.3   .97   5    23
southern          SO      Suan Chin          5.1   .95   4    15
northeast         NE      AM Main Jr.        5.1   .94   3    13
central           CT      Ann Stephens       5.7   .94   5    13
```

说明

打印所有包含数字 5，后跟一个句点，再跟一个任意字符的行。句点这个元字符通常代表单个字符，除非它被反斜杠转义。句点被转义后就不再是特殊的元字符，而只代表本身，即一个句号。

范例 4-12

```
% grep '\.5' datafile
north             NO       Margot Weber          4.5   .89    5   9
```

说明

打印所有包含表达式“.5”的行。

范例 4-13

```
% grep '^[we]' datafile
western          WE      Sharon Gray         5.3    .97    5   23
eastern          EA      TB Savage           4.4    .84    5   20
```

说明

打印所有以字母 w 或 e 开头的行。脱字符(^)是句首定位符，方括号中任何一个字符都可以被匹配。

范例 4-14

```
% grep '[^0-9]' datafile
northwest        NW     Charles Main        3.0    .98    3    34
western          WE     Sharon Gray         5.3    .97    5    23
southwest        SW     Lewis Dalsass       2.7    .8     2    18
southern         SO     Suan Chin           5.1    .95    4    15
southeast        SE     Patricia Hemenway   4.0    .7     4    17
eastern          EA     TB Savage           4.4    .84    5    20
northeast        NE     AM Main Jr.         5.1    .94    3    13
north            NO     Margot Weber        4.5    .89    5    9
central          CT     Ann Stephens        5.7    .94    5    13
```

说明

打印包含非数字字符的所有行。由于每一行都至少有一个非数字字符，因此所有行都被打印(参见表 4-2 中的-v 选项)。

范例 4-15

```
% grep '[A-Z][A-Z] [A-Z]' datafile
eastern         EA      TB Savage           4.4    .84    5    20
northeast       NE      AM Main Jr.         5.1    .94    3    13
```

说明

打印所有包含两个大写字符、后跟一个空格和一个大写字符的行，例如 TB Savage 和 AM Main。

范例 4-16

```
% grep 'ss* ' datafile
northwest       NW      Charles Main          3.0   .98     3    34
southwest       SW      Lewis Dalsass         2.7   .8      2    18
```

说明

打印所有包含一个 s、后跟 0 个或多个连着的 s 和一个空格的文本行。这个例子中，grep 找到了 Charles 和 Dalsass。



% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 4-17

```
% grep '[a-z]\{9\}' datafile
northwest       NW    Charles Main         3.0   .98   3   34
southwest       SW    Lewis Dalsass        2.7   .8    2   18
southeast       SE    Patricia Hemenway    4.0   .7    4   17
northeast       NE    AM Main Jr.          5.1   .94   3   13
```

说明

打印所有出现至少 9 个小写字母连在一起的行，例如，northwest、southwest、southeast 和 northeast。

范例 4-18

```
% grep '\(3\)\.[0-9].*\1    *\1' datafile
northwest       NW    Charles Main       3.0   .98    3    34
```

说明

如果某个文本行包含一个 3 后面跟一个句点和一个数字，再任意多个字符(.*)，然后跟一个 3 和任意多个制表符，再接一个 3，则打印该行。由于数字 3 被括在圆括号中，\(3\)，以后就可以用\1 来引用它。\1 代表被\(\)标记的第一个表达式。

范例 4-19

```
% grep '\<north' datafile
northwest       NW    Charles Main     3.0   .98   3   34
northeast       NE    AM Main Jr.      5.1   .94   3   13
north           NO    Margot Weber     4.5   .89   5    9
```

说明

打印所有包含以 north 开头的单词的行。“\<”是词首定位符。

范例 4-20

```
% grep '\<north\>' datafile
north           NO     Margot Weber     4.5   .89    5    9
```

说明

打印所有含单词 north 的行。“\<” 是词首定位符，“\>” 则是词尾定位符。

范例 4-21

```
% grep '\<[a-z].*n\>' datafile
northwest       NW       Charles Main       3.0    .98    3    34
western         WE       Sharon Gray        5.3    .97    5    23
southern        SO       Suan Chin          5.1    .95    4    15
eastern         EA       TB Savage          4.4    .84    5    20
northeast       NE       AM Main Jr.        5.1    .94    3    13
central         CT       Ann Stephens       5.7    .94    5    13
```

说明

打印所有包含以小写字母开头，以 n 结尾，中间由任意多个字符组成的单词的行。注意符号.*，它代表任意字符，包括空格。

4.3 grep 的选项

grep 提供了多个可用来控制其行为的选项。各种 UNIX 版本上 grep 的选项并不完全一样，因此一定要在手册中查找完整的 grep 选项清单。

本节中的示例使用下面的 datefile，为了阅读方便，该文件将会重复地出现。

% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 4-22

```
% grep -n '^south' datafile
3:southwest       SW       Lewis Dalsass          2.7    .8     2    18
4:southern        SO       Suan Chin              5.1    .95    4    15
5:southeast       SE       Patricia Hemenway      4.0    .7     4    17
```

说明

选项-n 在找到指定模式的行前面加上其行号再一并输出。



% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 4-23

```
% grep -i 'pat' datafile
southeast          SE      Patricia Hemenway      4.0   .7    4   17
```

说明

选项-i 关闭大小写敏感性。表达式 pat 包含任意大小写的组合都符合。

范例 4-24

```
% grep -v 'Suan Chin' datafile
northwest            NW     Charles Main          3.0    .98    3   34
western              WE     Sharon Gray           5.3    .97    5   23
southwest            SW     Lewis Dalsass         2.7    .8     2   18
southeast            SE     Patricia Hemenway     4.0    .7     4   17
eastern              EA     TB Savage             4.4    .84    5   20
northeast            NE     AM Main Jr.           5.1    .94    3   13
north                NO     Margot Weber          4.5    .89    5   9
central              CT     Ann Stephens          5.7    .94    5   13
```

说明

这个示例中，选项-v 打印所有不含模式 Suan Chin 的行。选项-v 可用来删除输入文件中的特定条目。如果要真正删除这些条目，就要把 grep 的输出重定向到一个临时文件，然后把这个临时文件的名字改成原始文件的名字，就像这样：

```
grep -v 'Suan Chin' datafile > temp
mv temp datafile
```

记住，必须使用临时文件来重定向源自 datafile 的输出。如果您从 datafile 重定向到 datafile，shell 就会“摧毁”datafile(请参见 1.6.6 节中的“重定向”)。

范例 4-25

```
% grep -l 'SE'  *
datafile
datebook
```

说明

选项-l 使 grep 只输出包含模式的文件名，而不输出文本行。

范例 4-26

```
% grep -c 'west' datafile
3
```

说明

选项-c 让 grep 打印出含有模式的行的数目。这个数字并不代表模式的出现次数。例如，即使 west 在某行中出现了 3 次，这行也只计一次。

范例 4-27

```
% grep  -w  'north'  datafile
north            NO      Margot Weber           4.5   .89    5     9
```

说明

选项-w 使 grep 只查找作为一个词，而不是词的一部分出现的模式[③]。这条命令只打印包含词 north 的行，而不打印那些在 northwest、northeast 等中出现 north 的行。

范例 4-28

```
% echo  $LOGNAME
lewis
% grep -i "$LOGNAME" datafile
southwest            SW       Lewis Dalsass          2.7   .8    2    18
```

说明

打印 shell 的环境变量 LOGNAME 的值，这个值记录了用户的登录名。即使变量被括在双引号之间，也会被 shell 展开，而且，如果有变量的值中有多个词时，用于分隔它们的空白符也可免受 shell 的解释。如果用的是单括号，就不会发生变量替换，也就是说，打印结果是$LOGNAME。

4.4 grep 与管道

grep 的输入并不一定都是文件，它也常常从管道读取输入。

范例 4-29

```
% ls -l
drwxrwxrwx  2  ellie    2441 Jan 6 12:34   dir1
-rw-r--r--  1  ellie    1538 Jan 2 15:50   file1
-rw-r--r--  1  ellie    1539 Jan 3 13:36   file2
drwxrwxrwx  2  ellie    2341 Jan 6 12:34   grades
```

③ 这里所说的词是指一个字母或数字字符序列，它始于行首或紧跟在空白符后，以空白符、标点或换行符结束。




```
% ls -l | grep '^d'
drwxrwxrwx  2  ellie    2441 Jan 6 12:34    dir1
drwxrwxrwx  2  ellie    2341 Jan 6 12:34    grades
```

说明

ls 命令的输出通过管道传给 grep。输出结果中以字母 d 开头的所有行都被打印出来，也就是说，所有目录被打印出来。

grep 回顾

表 4-3 中给出了 grep 命令的一些示例，并描述了它们执行的操作。

表 4-3　grep 回顾

grep 命令	命令执行的操作
grep '\<Tom\>' file	打印包含词 Tom 的行
grep 'Tom Savage' file	打印包含 Tom Savage 的行
grep '^Tommy' file	打印以 Tommy 开头的行
grep '\.bak$' file	打印以.bak 结尾的行。单引号保护了美元符号($)，使之不被 shell 解释
grep '[Pp]yramid' *	打印当前工作目录下所有文件中包含 pyramid 或 Pyramid 的行
grep '[A-Z]' file	打印所有至少包含一个大写字母的行
grep '[0-9]' file	打印所有至少包含一个数字的行
grep '[A-Z]...[0-9]' file	打印包含这样一个模式的行，该模式以大写字母开头、数字结尾、共有五个字符
grep –w '[tT]est' files	打印包含词 Test 或 test 的行
grep –s 'Mark Todd' file	查找包含 Mark Todd 的行，但不打印找到的行。可用于检查 grep 的退出状态
grep –v 'Mary' file	打印所有不含 Mary 的行
grep –i 'sam' file	打印所有包含 sam 的行，sam 的各种大小写形式都可以(如：SAM、sam、Sam、sAm)
grep –l 'Dear Boss' *	列出所有包含 Dear Boss 的文件名
grep –n 'Tom' file	在每个匹配行前面加上行号
grep "$name" file	展开变量 name 的值，打印包含该值的所有行。必须用双引号
grep '$5' file	打印包含$5 的行。必须用单引号
ps –ef | grep '^ *user1'	把 ps –ef 的输出经管道发给 grep，在行首查找 user1，即使 user1 前面有多个空格也行

4.5　egrep(扩展的 grep)

使用 egrep 的主要好处是它在 grep 提供的正则表达式元字符集的基础上增加了更多的

元字符(参见表 4-4)。但是，egrep 不允许使用\(\)和\{\}(如果使用的是 Linux 系统，请参考 GNU 的 grep –E 命令)。

表 4-4　egrep 使用的正则表达式元字符

<table>
<tr><th>元字符</th><th>功　能</th><th>示　例</th><th>匹 配 对 象</th></tr>
<tr><td>^</td><td>行首定位符</td><td>'^love'</td><td>匹配所有以 love 开头的行</td></tr>
<tr><td>$</td><td>行尾定位符</td><td>'love$'</td><td>匹配所有以 love 结尾的行</td></tr>
<tr><td>.</td><td>匹配一个字符</td><td>'l..e'</td><td>匹配包含一个 l，后跟两个字符，再跟一个 e 的行</td></tr>
<tr><td>*</td><td>匹配零个或多个前导字符</td><td>' *love'</td><td>匹配包含跟在零个或多个空格后的模式 love 的行</td></tr>
<tr><td>[ ]</td><td>匹配一组字符中任一个</td><td>'[Ll]ove'</td><td>匹配包含 love 或 Love 的行</td></tr>
<tr><td>[^]</td><td>匹配不在指定字符组内的字符</td><td>'[^A-KM-Z]ove'</td><td>匹配包含 ove、但 ove 前面那个字符既不在 A 到 K 之间、也不在 M 到 Z 之间的行</td></tr>
<tr><td colspan="4">egrep 新增的元字符</td></tr>
<tr><td>+</td><td>匹配一个或多个加号前的字符</td><td>'[a-z]+ove'</td><td>匹配一个或多个小写字母后跟 ove 的字符串。将找出 move、approve、love、behoove 等</td></tr>
<tr><td>?</td><td>匹配零个或一个前导字符</td><td>'lo?ve'</td><td>匹配 l 后跟一个或零个字母 o 以及 ve 的字符串。将找到 love 或 lve</td></tr>
<tr><td>a|b</td><td>匹配 a 或 b</td><td>'love|hate'</td><td>匹配 love 和 hate 这两个表达式之一</td></tr>
<tr><td>( )</td><td>字符组</td><td>'love(able|ly)(ov)+'</td><td>匹配 lovable 或 lovely
匹配 ov 的一次或多次出现</td></tr>
</table>

4.5.1　egrep 示例

下面这些示例只介绍 egrep 如何使用扩展集中那部分新的正则表达式元字符。之前给出的 grep 示例已经说明了标准元字符的用法，和在 egrep 中的用法相同。egrep 所用的命令行参数也和 grep 的一样。

范例 4-30

```
% egrep 'NW|EA' datafile
northwest       NW      Charles Main        3.0     .98     3     34
eastern         EA      TB Savage           4.4     .84     5     20
```

说明

打印包含表达式 NW 或 EA 的行。

范例 4-31

```
% egrep '3+' datafile
northwest       NW      Charles Main        3.0     .98     3     34
western         WE      Sharon Gray         5.3     .97     5     23
northeast       NE      AM Main Jr.         5.1     .94     3     13
central         CT      Ann Stephens        5.7     .94     5     13
```



% cat datafile						
northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

说明

打印所有包含一个或多个数字3的行。

范例 4-32

```
% egrep '2\.?[0-9]' datafile
western         WE      Sharon Gray        5.3   .97   5   23
southwest       SW      Lewis Dalsass      2.7   .8    2   18
eastern         EA      TB Savage          4.4   .84   5   20
```

说明

打印所有包含数字2，后面跟零个或一个句点，再跟一个数字的行，将匹配2.5、25、29、2.3等。

范例 4-33

```
% egrep '(no)+' datafile
northwest       NW      Charles Main       3.0   .98   3   34
northeast       NE      AM Main Jr.        5.1   .94   3   13
north           NO      Margot Weber       4.5   .89   5    9
```

说明

打印连续出现一个或多个模式组no的行，将匹配no、nono、nononononono等。

范例 4-34

```
% egrep 'S(h|u)' datafile
western         WE      Sharon Gray        5.3   .97   5   23
southern        SO      Suan Chin          5.1   .95   4   15
```

说明

打印所有包含字母S，后跟h或u的行，将匹配**Sharon**和**Suan**。

范例 4-35

```
% egrep 'Sh|u' datafile
western         WE      Sharon Gray        5.3   .97   5   23
```

```
southern        SO    Suan Chin            5.1   .95   4   15
southwest       SW    Lewis Dalsass        2.7   .8    2   18
southeast       SE    Patricia Hemenway    4.0   .7    4   17
```

说明

打印所有包含表达式 Sh 或 u 的行。将匹配 Sharon 或 southern。

4.5.2 egrep 回顾

表 4-5 中给出了 egrep 命令的示例，并说明了它们所执行的操作。

表 4-5 egrep 回顾

命　令	命令执行的操作
egrep '^ +' file	打印以一个或多个空格开头的行
egrep '^ *' file	打印以零个或多个空格开头的行[④]
egrep ' (Tom|Dan) Savage' file	打印包含 Tom Savage 或 Dan Savage 的行
egrep '(ab)+ ' file	打印出现一个或多个 ab 的行
egrep '^X[0-9]?' file	打印以 X 开头，X 后面跟零个或一个数字的行
egrep 'fun\.$' *	打印当前工作目录下所有文件中以 fun.结尾的行[④]
egrep '[A-Z]+' file	打印包含一个或多个大写字母的行
egrep '[0-9]' file	打印包含数字的行[④]
egrep '[A-Z]...[0-9]' file	打印包含这样一个模式的行，该模式以大写字母开头、数字结尾、中间 3 个任意字符，共计 5 个字符[④]
egrep '[tT]est' files	打印包含 Test 或 test 的行[④]
egrep ' (Susan | Jean)Doe' file	打印包含 Susan Doe 或 Jean Doe 的行[④]
egrep –v 'Mary' file	打印所有不含 Mary 的行[④]
egrep –i 'sam' file	打印所有包含 sam 的行，sam 的各种大小写形式都可以(如：SAM、sam、Sam、sAm)[④]
egrep –l 'Dear Boss' *	列出所有包含 Dear Boss 的文件名[④]
egrep –n 'Tom' file	在每个匹配行前面加上行号[④]
egrep –s '$name' file	展开变量 name 的值，查找它。但是不打印结果。可用来检查 egrep 的退出状态[④]

4.6 fgrep(固定的 grep 或快速的 grep)

fgrep 命令的运行方式与 grep 类似，但它不对任何正则表达式元字符做特殊处理。所

④ egrep 和 grep 对该模式的处理方式相同。



有字符都只代表它们自己：脱字符就是脱字符，美元符就是美元符，全部如此(如果使用的是 Linux 系统，请参考 GNU 的 grep –F 命令)。

范例 4-36

```
fgrep '[A-Z]****[0-9]..$5.00'  file
```

说明

查找文件中所有包含字符串[A-Z]****[0-9]..$5.00 的行。所有字符都代表它们本身，没有什么特殊含义。

4.7 Linux 与 GNU grep

Linux 使用 GNU 版本的 grep，其功能大部分与 grep 相同，只是有些方面做得更好。除 POSIX 字符(参见表 4-7 和表 4-8)外，还包含了很多新的选项，如-G、-E、-F 和-P 等使用普通 grep 的选项，另外还有 egrep 和 fgrep[5]的功能。



基本正则表达式与扩展正则表达式

GNU grep 命令支持与 UNIX grep 相同的正则表达式元字符(参见表 4-7)。同时，修改了部分元字符(参见表 4-8)搜索与行显示方式。例如，可以提供选项来关闭大小写敏感、显示行号、显示文件名等。

有两种版本的正则表达式元字符：基本元字符和扩展元字符。标准版本的 GNU grep(grep –G)使用基本集(参见表 4-7)，egrep(或 grep -E)使用扩展集(参见表 4-8)。对 GNU grep，两种版本均可用。基本集包括^, $, ., *, [ ], [^], \< \>, 和\(\)。

另外，GNU grep 识别\b、\w 和\W 以及一种新类别：POSIX 元字符(参见表 4-9)。

以-E 选项使用 GNU grep，则扩展集(egrep)可用。即使没有-E 选项，默认设置的标准 grep 也可以使用扩展集中的元字符。仅需要对这些元字符前置一个反斜线[6]。例如，扩展集元字符为

```
?, +, { }, |, ( )
```

扩展集中的元字符对标准 grep 来讲没有任何特殊意义，除非在它们前面以如下方式加上一个反斜线：

```
\?, \+, \{, \|, \(, \)
```

GNU grep 的使用格式参见表 4-6。

[5] 递归使用 grep，参见附录 A 中关于 GNU rgrep 和 xargs 的相关内容。

[6] 在任何版本的 grep 中，可以使用反斜线来引用元字符以关闭它的特殊含义。

表 4-6 GNU grep

格　式	含　义
grep 'pattern' filename(s)	基本的 RE 元字符(默认)
grep –G 'pattern' filename(s)	含义同上(默认)
grep –E 'pattern' filename(s)	扩展 RE 元字符
grep –F 'pattern' filename	非 RE 元字符
grep –P 'pattern' filename	将模式解释为 Perl RE

表 4-7 GNU 版本 grep 的正则表达式元字符基本集

元字符	功　能	示例	匹 配 内 容
^	行首定位符	^love	匹配所有以 love 开头的行
$	行尾定位符	love$	匹配所有以 love 结尾的行
.	匹配单个字符	l..e	匹配包含一个 l，后接两个字符，再接一个 e 的行
*	匹配零个或多个字符	*love	匹配以零个或多个空格开始，后跟 love 模式的行
[ ]	匹配集合中的一个字符	[Ll]ove	匹配包含 love 或 Love 的行
[^]	匹配非集合中的一个字符	[^A–K]ove	匹配包含不以 A 至 K 之间的某个字符开头，后接 ove 的行
\<[7]	词首定位符	\<love	匹配包含以 love 开头的词的行
\>	词尾定位符	love\>	匹配包含以 love 结尾的词的行
\(..\)[8]	标签匹配字符	\(love\)able	寄存器中以标签标记的部分，以数字 1 记录。将来引用时，用\1 重复该模式。最多可以使用九个标签，模式最左侧部分为第一个标签。例如，模式 love 保存在寄存器 1 中，将来以\1 进行引用。
x\{m\}	字符 x 重复 m 次	o\{5\}	匹配字母 o 出现 5 次的行
x\{m,\}	字符 x 重复至少 m 次	o\{5,\}	匹配字母 o 出现至少 5 次的行
x\{m,n\}[9]	字符 x 重复 m 到 n 次	o\{5,10\}	匹配字母 o 出现 5~10 次的行
\w	所有字母与数字，称为字符[a-zA-Z0-9_]	l\w*e	匹配一个 l 后跟零个或多个字符，最后接一个 e
\W	所有字母与数字之外的字符，称为非字符[^a-zA-Z0-9_]	love\W+	匹配 love 后接一个或多个非字符(.,？等)
\b	词边界	\blove\b	仅匹配 love 这个单词

⑦ 除非使用反斜线，否则即使是使用 grep –E 和 GNU egrep 也不会工作。而在 UNIX egrep 上无论如何也不会工作。

⑧ 这些元字符确属扩展集的一部分。之所以放在这里是因为它们在使用反斜线的情况下能够在 UNIX grep 和 GNU regular grep 上工作。它们根本不会在 UNIX egrep 上工作。

⑨ 所有版本的 UNIX 及所有模式匹配工具均不支持元字符\{ \}，通常在 vi 和 grep 中使用它们。它们根本不能在 UNIX egrep 上工作。

表 4-8 egrep 与 grep –E 使用的扩展集

元字符	功 能	实 例	匹配内容
+	匹配一个或多个前导字符	[a–z]+ove	匹配一个或多个小写字母，后跟 ove。如 move，approve，love，behoove 等
?	匹配零个或一个前导字符	lo?ve	匹配一个 l 后跟一个或零个字母 o 的模式，如 love 或 lve
a|b|c	匹配 a 或 b 或 c	love|hate	匹配其中的一个表达式，love 或 hate
( )	组字符	love(able|rs) (ov)+	匹配 loveable 或 lovers 匹配一个或多个连续的 ov
(..) (...) \1 \2[⑩]	标签匹配字符	\(love\)ing	寄存器中以标签标记的部分，以数字 1 记录。将来引用时，用\1 重复该模式。最多可以使用 9 个标签，模式最左侧部分为第一个标签。例如，模式 love 保存在寄存器 1 中，将来以\1 进行引用
x{m}	字符 x 重复 m 次	o\{5\}	匹配字母 o 出现 5 次的行
x{m,}	字符 x 重复至少 m 次	o\{5,\}	匹配字母 o 出现至少 5 次的行
x{m,n}[⑪]	字符 x 重复 m 到 n 次	o\{5,10\}	匹配字母 o 出现 5~10 次的行

POSIX 类 POSIX(可移植操作系统接口，the Portable Operating System Interface)是一个保证程序能够跨操作系统移植的工业标准。为了实现可移植性，POSIX 认可不同的国家和地区在字符编码、货币表示以及时间和日期的表示方式上的不同。为处理不同类型的字符，POSIX 在基本正则表达式与扩展正则表达式的基础上加入了表 4-9 所示的括号字符类的字符。

这种字符类，例如，[:alnum:]是 A–Za–z0–9 的另一种表达方式。为使用这种字符类，它必须使用另外一对括号进行引用以将其标识为一个正则表达式。例如，A–Za–z0–9 本身并不是正则表达式，但[A–Za–z0–9]是。同样地，[:alnum:] 应写作[[:alnum:]]。使用第一种形式[A–Za–z0–9]与使用括号形式的[[:alnum:]]之间的差别在于第一种形式依赖于 ASCII 字符编码，而第二种形式可以在该类中表示来自其他语言的字符，如瑞典语中的 ring 字符和德语中的元音变音字符(umlaut)。

表 4-9 括号字符类

括 号 类	含 义
[:alnum:]	字母与数字两种字符
[:alpha:]	字母字符
[:cntrl:]	控制字符

⑩ 这种标签与引用在 UNIX egrep 上不能工作。

⑪ 所有版本的 UNIX 及所有模式匹配工具均不支持元字符\{ \}；通常在 vi 和 grep 中使用它们。它们根本不能在 UNIX egrep 上工作。

(续表)

括号类	含义
[:digit:]	数字字符
[:graph:]	非空字符(不包含空格、控制字符等)
[:lower:]	小写字母
[:print:]	与[:graph:]类似，但包含空格字符
[:punct:]	标点字符
[:space:]	所有的空白字符(换行符，空格符，制表符)
[:upper:]	大写字母
[:xdigit:]	十六进制数字字符(0-9a-fA-F)

范例 4-37

```
1  % grep '[[:space:]]\.[[:digit:]][[:space:]]' datafile
   southwest   SW    Lewis Dalsass          2.7    .8   2    18
   southeast   SE    Patricia Hemenway      4.0    .7   4    17

2  % grep -E '[[:space:]]\.[[:digit:]][[:space:]]' datafile
   southwest   SW    Lewis Dalsass          2.7    .8   2    18
   southeast   SE    Patricia Hemenway      4.0    .7   4    17

3  % egrep '[[:space:]]\.[[:digit:]][[:space:]]' datafile
   southwest   SW    Lewis Dalsass          2.7    .8   2    18
   southeast   SE    Patricia Hemenway      4.0    .7   4    17
```

说明

1，2，3 适用于所有 grep(而非 fgrep)的 Linux 变体，支持 POSIX 括号字符类集合。每个例子中，grep 搜索一个空格字符、一个句点、一个数字和另一个空格字符。

4.8 带正则表达式的 GNU 基本 grep(grep -G)

基本 grep 将它的模式解释为基本正则表达式。本节中所有的 UNIX 基本 grep 的例子同样也适用于 GNU 版本的基本 grep，以及 grep –G 或 grep '– –basic–regexp'。

下面范例中使用的元字符不能在基本 UNIX grep 中找到。本节中的例子使用下面的 datafile 文件。

范例 4-38

```
% grep NW datafile          或
% grep -G NW datafile
northwest        NW      Charles Main        3.0   .98   3    34
```



% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

说明

打印文件 datafile 中所有包含正则表达式 NW 的行。

范例 4-39

```
% grep '^n\w*\W' datafile
northwest          NW     Charles Main          3.0   .98   3    34
northeast          NE     AM Main Jr.           5.1   .94   3    13
```

说明

打印以一个 n 开始，后面跟零个或多个的字母数字混合的字符[a-zA-Z0-9_]，后面跟一个非字母数字的字符[^a-zA-Z0-9_]。\w 和\W 是 grep 的 GNU 变体中的标准单词元字符。

范例 4-40

```
% grep '\bnorth\b' datafile
north              NO     Margot Weber          4.5   .89   5    9
```

说明

打印包含词 north 的行。\b 是一个词分界符。在所有 grep 的 GNU 变体上它可以用来代替词定位符(\< \>)。

4.9 grep –E 或 egrep(GNU 扩展 grep)

使用扩展 grep 的主要优势是有附加的正则表达式元字符(参见表 4-10)加入到基本集中。通过-E 扩展，GNU grep 可以使用这些新的元字符。

表 4-10　egrep 的正则表达式元字符

元字符	功　能	实　例	匹 配 内 容
^	行首定位符	^love	匹配所有以 love 开头的行
$	行尾定位符	love$	匹配所有以 love 结尾的行
.	匹配单个字符	l..e	匹配包含一个 l，后接两个字符，再接一个 e 的行
*	匹配零个或多个字符	*love	匹配以零个或多个空格开始，后跟 love 模式的行
[ ]	匹配集合中的一个字符	[Ll]ove	匹配包含 love 或 Love 的行
[^]	匹配非集合中的一个　字符	[^A–KM-Z]ove	匹配不包含以 A 至 K 或 M 至 Z 之间的某个字符开头，后接 ove 的行
grep –E 或 egrep 新增元字符			
+	匹配一个或多个前导字符	[a–z]+ove	匹配一个或多个小写字母，后跟 ove。如 move，approve，love，behoove 等
?	匹配零个或一个前导字符	lo?ve	匹配一个 l 后跟一个或零个字母 o，如 love 或 lve
a|b	匹配 a 或 b	love|hate	匹配其中的一个表达式，love 或 hate
( )	组字符	love(able|ly) (ov)+	匹配 loveable 或 lovely。 匹配一个或多个 ov 模式
x\{m\}	字符 x 重复 m 次	o\{5\}	匹配字母 o 出现 5 次的行
x\{m,\}	字符 x 重复至少 m 次	o\{5,\}	匹配字母 o 出现至少 5 次的行
x\{m,n\}[12]	字符 x 重复 m~n 次	o\{5,10\}	匹配字母 o 出现 5~10 次的行
\w	所有字母与数字，称为词字符[a-zA-Z0-9_]	l\w*e	匹配一个 l 后跟零个或多个词字符，最后是一个 e
\W	所有字母与数字之外的字符，称为非词字符[^a-zA-Z0-9_]	\W\w*	匹配一个非词字符(\W)，后跟零个或多个词字符(\w)
\b	词边界	\blove\b	仅匹配词 love

4.9.1　grep –E 和 egrep 实例

以下的例子示意了 grep –E 和 egrep 使用扩展集正则表达式元字符的方式。首先以一个 grep 实例示意标准元字符的用法，这些标准元字符同样适用于 egrep。基本 GNU grep(grep -G) 可以使用任何附加的元字符，只需在特殊元字符前加上一个反斜线。

[12] 所有版本的 UNIX 及所有模式匹配工具均不支持元字符{ }；通常在 vi 和 grep 中使用它们。它们根本不能在 UNIX egrep 上工作。



以下的范例显示了 grep 三种变体是如何完成同样的任务的。

本节所有的范例均使用下面的 datafile，为方便查阅，它将周期地重复出现。

```
% cat datafile
northwest    NW    Charles Main       3.0   .98   3   34
western      WE    Sharon Gray        5.3   .97   5   23
southwest    SW    Lewis Dalsass      2.7   .8    2   18
southern     SO    Suan Chin          5.1   .95   4   15
southeast    SE    Patricia Hemenway  4.0   .7    4   17
eastern      EA    TB Savage          4.4   .84   5   20
northeast    NE    AM Main Jr.        5.1   .94   3   13
north        NO    Margot Weber       4.5   .89   5   9
central      CT    Ann Stephens       5.7   .94   5   13
```

范例 4-41

```
1  % egrep 'NW|EA' datafile
   northwest        NW      Charles Main       3.0   .98   3   34
   eastern          EA      TB Savage          4.4   .84   5   20

2  % grep -E 'NW|EA' datafile
   northwest        NW      Charles Main       3.0   .98   3   34
   eastern          EA      TB Savage          4.4   .84   5   20

3  % grep 'NW|EA' datafile

4  % grep 'NW\|EA' datafile
   northwest        NW      Charles Main       3.0   .98   3   34
   eastern          EA       TB Savage         4.4   .84   5   20
```

说明

1. 打印包含表达式 NW 或 EA 的行。本例中使用的是 egrep。如果没有 GNU 版本的 grep，则用 egrep 代替。

2. 本例中，使用了带-E 选项的 GNU grep 以将扩展元字符包括在内。与 egrep 相同。

3. 标准的 grep 通常并不支持扩展正则表达式；竖线是扩展正则表达式用作间隔的元字符。标准 grep 不能识别它，因此搜索显式模式'NW|EA'，没有匹配，没有打印输出。

4. 如果在 GNU 标准 grep(grep -G)的元字符前加上一个反斜线，则与 egrep 和 grep –E 一样，该元字符将被解释为一个扩展正则表达式。

范例 4-42

```
% egrep '3+' datafile
% grep -E '3+' datafile
% grep '3\+' datafile
northwest        NW      Charles Main       3.0   .98   3   34
western          WE      Sharon Gray        5.3   .97   5   23
northeast        NE      AM Main Jr.        5.1   .94   3   13
central          CT      Ann Stephens       5.7   .94   5   13
```

% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	53	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

说明

打印所有包含一个或多个 3 的行。

范例 4-43

```
% egrep '2\.?[0-9]' datafile
% grep -E '2\.?[0-9]' datafile
% grep '2\.\?[0-9]' datafile
western         WE    Sharon Gray      5.3   .97   5   23
southwest       SW    Lewis Dalsass    2.7   .8    2   18
eastern         EA    TB Savage        4.4   .84   5   20
```

说明

打印所有包含一个 2，后跟 0 个或 1 个句点，再接一个 0~9 之间的数的行。

范例 4-44

```
% egrep '(no)+' datafile
% grep -E '(no)+' datafile
% grep '\(no\)\+' datafile
northwest       NW    Charles Main      3.0   .98   3   34
northeast       NE    AM Main Jr.       5.1   .94   3   13
north           NO    Margot Weber      4.5   .89   5   9
```

说明

打印包含一个或多个模式组 no 的行。

范例 4-45

```
% grep -E '\w+\W+[ABC]' datafile
northwest       NW    Charles Main       3.0   .98   3   34
southern        SO    Suan Chin          5.1   .95   4   15
northeast       NE    AM Main Jr.        5.1   .94   3   13
central         CT    Ann Stephens       5.7   .94   5   13
```

说明

打印所有包含一个或多个字母数字词字符(\w+)，后跟一个或多个非字母数字词字符(\W+)，再接上集合 ABC 中一个字母的行。



范例 4-46

```
% egrep 'S(h|u)' datafile
% grep -E 'S(h|u)' datafile
% grep 'S\(h\|u\)' datafile
western         WE      Sharon Gray      5.3   .97   5   23
southern        SO      Suan Chin        5.1   .95   4   15
```

说明

打印所有包含一个 S 后跟一个 h 或 u(如 Sh 或 Su)的行。

范例 4-47

```
% egrep 'Sh|u' datafile
% grep -E 'Sh|u' datafile
% grep 'Sh\|u' datafile
western         WE      Sharon Gray          5.3   .97   5   23
southern        SO      Suan Chin            5.1   .95   4   15
southwest       SW      Lewis Dalsass        2.7   .8    2   18
southeast       SE      Patricia Hemenway    4.0   .7    4   17
```

说明

打印所有包含表达式 Sh 或 u 的行。

4.9.2 grep 变体的不规则形式

Linux 支持的 GNU grep 变体几乎等同于它们在 UNIX 上的同名 grep，但也不是完全相同。例如，Solaris 或 BSD UNIX 上的 egrep 版本不支持 3 种元字符：用于重复的\{ \}、标签字符\(\)和词定位符\< \>。在 Linux 系统上，grep 和 grep –E 可以识别这 3 种元字符，但 egrep 不能识别\< \>。下面的范例示意了它们之间的差别，假定是在 UNIX 系统而不是 Linux 系统上运行 bash 或 tcsh，并在 shell 脚本中使用 grep 及 grep 族。

本节所有的范例使用下面的 datafile，为方便，它将周期地重复出现。

```
% cat datafile
northwest    NW    Charles Main         3.0   .98   3   34
western      WE    Sharon Gray          53    .97   5   23
southwest    SW    Lewis Dalsass        2.7   .8    2   18
southern     SO    Suan Chin            5.1   .95   4   15
southeast    SE    Patricia Hemenway    4.0   .7    4   17
eastern      EA    TB Savage            4.4   .84   5   20
northeast    NE    AM Main Jr.          5.1   .94   3   13
north        NO    Margot Weber         4.5   .89   5   9
central      CT    Ann Stephens         5.7   .94   5   13
```

范例 4-48

```
(Linux GNU grep)
1  % grep '<north>' datafile    # 必须使用反斜杠
2  % grep '\<north\>' datafile
   north            NO      Margot Weber          4.5   .89    5    9
3  % grep -E '\<north\>' datafile
   north            NO      Margot Weber          4.5   .89    5    9
4  % egrep '\<north\>' datafile
   north            NO      Margot Weber          4.5   .89    5    9
(Solaris egrep)
5  % egrep '\<north\>' datafile
   <no output; not recognized>
```

说明

1. 无论使用哪种 grep 变体，都必须在词定位符元字符前加上一个反斜线。

2. 这一次，grep 搜索一个以 north 开头并以之结尾的词。\<代表词首定位符，\>代表词尾定位符。

3. 带-E 选项的 grep 也能够识别词定位符。

4. GNU 版本的 egrep 可以识别词定位符。

5. 当使用 Solaris(SVR4)时，egrep 不能将词定位符识别为正则表达式元字符。

范例 4-49

```
(Linux GNU grep)
1  % grep 'w(es)t.*\1' datafile
   grep: Invalid back reference
2  % grep 'w\(es\)t.*\1' datafile
   northwest         NW      Charles Main          3.0   .98    3    34
3  % grep -E 'w(es)t.*\1' datafile
   northwest         NW      Charles Main          3.0   .98    3    34
4  % egrep 'w(es)t.*\1' datafile
   northwest         NW      Charles Main          3.0   .98    3    34
(Solaris egrep)
5  % egrep 'w(es)t.*\1' datafile
   <no output; not recognized>
```

说明

1. 当使用标准 grep 时，扩展元字符( )前必须加上反斜线，否则会出错。

2. 如果正则表达式 w\(es\)t 匹配成功，则模式 es 被存储在内存寄存器 1 中。该表达式含义是：如果找到了 west，则标记并保存模式 es，接着，搜索任意数目的字符后再次匹配 es(\1)的行，并打印该行。Charles 中的 es 在后向引用时被匹配成功。

3. 与前一个例子相同，不同之处在于带-E 选项的 grep 不用在( )前加上一个反斜线。

4. GNU egrep 也使用不带反斜线的扩展元字符( )。

5. 在 Solaris 系统上，egrep 不能识别任何形式的标签及后向引用。



% cat datafile						
northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 4-50

```
(Linux GNU grep)
1  % grep '\.[0-9]\{2\}[^0-9]' datafile
   northwest       NW      Charles Main    3.0     .98     3       34
   western         WE      Sharon Gray     5.3     .97     5       23
   southern        SO      Suan Chin       5.1     .95     4       15
   eastern         EA      TB Savage       4.4     .84     5       20
   northeast       NE      AM Main Jr.     5.1     .94     3       13
   north           NO      Margot Weber    4.5     .89     5        9
   central         CT      Ann Stephens    5.7     .94     5       13

2  % grep -E '\.[0-9]{2}[^0-9]' datafile
   northwest       NW      Charles Main    3.0     .98     3       34
   western         WE      Sharon Gray     5.3     .97     5       23
   southern        SO      Suan Chin       5.1     .95     4       15
   eastern         EA      TB Savage       4.4     .84     5       20
   northeast       NE      AM Main Jr.     5.1     .94     3       13
   north           NO      Margot Weber    4.5     .89     5        9
   central         CT      Ann Stephens    5.7     .94     5       13

3  % egrep  '\.[0-9]{2}[^0-9]' datafile
   northwest       NW      Charles Main    3.0     .98     3       34
   western         WE      Sharon Gray     5.3     .97     5       23
   southern        SO      Suan Chin       5.1     .95     4       15
   eastern         EA      TB Savage       4.4     .84     5       20
   northeast       NE      AM Main Jr.     5.1     .94     3       13
   north           NO      Margot Weber    4.5     .89     5        9
   central         CT      Ann Stephens    5.7     .94     5       13

   (Solaris egrep)
4  % egrep  '\.[0-9]{2}[^0-9]' datafile
    <no output; not recognized with or without backslashes>
```

说明

1. 扩展元字符{ }用于重复。GNU 与 UNIX 版本的标准 grep 并不对该扩展元字符集求

值，除非括号前加上反斜线。整个表达式含义是：搜索一个句点\.，后跟一个 0~9 之间的数字[0–9]，如果此模式恰好重复两次\{2\}，后跟一个非数字[^0–9]。

2. 扩展 grep，grep -E 使用重复元字符{2}，不必像上面例子那样在前面加一个反斜线。

3. 因为 GNU egrep 和 grep –E 功能相同，所以该命令产生的输出与前一个例子相同。

4. 这是标准的 UNIX 版本的 egrep。无论是否加上反斜线，它都不能将花括号识别为一个扩展元字符。

4.10 固定的 grep(grep –F 和 fgrep)

fgrep 命令的行为与 grep 的类似，但它不能够识别任何正则表达式元字符的特殊意义。所有字符仅能代表它们自身。脱字号就是脱字号，美元符号就是美元符号，以此类推。带 -F 选项的 GNU grep 的行为与 fgrep 的完全一致。

范例 4-51

```
% fgrep '[A-Z]****[0-9]..$5.00' file     或
% grep -F '[A-Z]****[0-9]..$5.00' file
```

说明

查找文件中所有包含字符串[A–Z]****[0–9]..$5.00 的行。所有字符只代表它们本身，不表示任何特殊含义。

4.11 递归的 grep(rgrep,grep -R)

与 grep 族的成员不同，Linux 上的 rgrep 可以沿一个目录树递归而下。rgrep 有许多命令行选项并支持与标准 grep(grep -R)相同的元字符。附录 A 给出了 rgrep 的完整描述，也可键入 rgrep-?以获得在线帮助(UNIX 标准版本上不支持)。

范例 4-52

```
% grep -r 'Tom' ./dir
% rgrep 'Tom' ./dir
```

说明

递归地搜索./dir 目录下包含字符串 Tom 的所有文件。

4.12 带选项的 GNU grep

grep 命令有许多可以控制其行为的选项。GNU 版本的 grep 又加入了许多新选项，同时对原来的一些选项提供了另外的选择方式。GNU grep 选项在 grep 所有不同的变体上都



能够工作，包括 grep –G，-E 和-F，如表 4-11 所示。

表 4-11　所有变体(-G,-E 和-F)均可用的 GNU grep 选项

选　项	作　用
-#- (# 是一个用来代表整数值的符号)	将匹配行前后#行的内容一同打印出来；也就是说，grep –2 pattern filename 将导致 grep 打印匹配行及匹配行的前两行和后两行
-A #, --after-context=#	打印匹配行后面#行的内容；也就是说，匹配行及它后面指定的#行内容
-B #, --before-context=#	打印匹配行前面#行的内容；也就是说，匹配行及它前面指定的#行内容
-C #, --context=#	等价于 - 2 选项。打印匹配行的前两行和后两行
-V, --version	打印 grep 版本信息，版本信息应当包含在所有的 bug 报告中
-a, --text, --binary-files=text	将二进制文件当作文本文件处理
-b, --byte-offset	在输出的每行前显示偏移字节数
-c, --count	为每个输入文件打印成功匹配的行数。-v 则打印一些未匹配的行数
-D action, --devices=action	如果输入文件为一个设备，如套接字或管道。则 action 默认从该设备读，就如同读一个普通文件一样。如果 action 为 skip，则该设备被忽略
-e PATTERN, --regexp=PATTERN	使用字面 PATTERN 作为模式；这对保护以-开头的模式非常有帮助
-f FILE, --file=FILE	从 FILE 中获得模式，每行一个。空文件包含 0 个模式，因此什么也不能匹配
--help--	显示有关 grep 命令行选项及错误报告地址的帮助信息，然后退出
-h, --no-filename	当搜索多个文件时，禁止输出文件名前缀
-i, --ignore-case	忽略模式和输入文件的大小写区别
-L, --files-without-match	仅打印所有未能匹配模式的文件名
-l, --files-with-matches	仅打印所有正确匹配模式的文件名
-m #, --max-count-#	如果文件是标准输入或正规文件，在找到指定数量(#)的匹配行后停止读文件
-n, --line-number	在匹配成功的输出行前加上行号作为前缀
-q, --quiet	禁止正规输出。可用来替代-n
-r, -R, --recursive, --directories=recurse	对列出的目录，递归地读并处理这些目录中的所有文件；也就是指该目录下的所有目录
-s, --silent	禁止显示文件不存在或文件不可读的错误信息
-v, --revert-match	转换匹配性质，选择非匹配行

(续表)

选　　项	作　　用
-w, --word-regexp	仅选择包含词匹配的行。匹配词边界上包含字母、数字和下划线的字符串
-x, --line-regexp	仅选择精确匹配整行的那些匹配
-y	与已废除的-i 同义
-U, --binary	将文件作为二进制文件处理。仅有 MS-DOS 和 MS-Windows 支持该选项
-u, --unix-byte-offsets	报告 UNIX 风格的字节偏移。这个选项仅在同时使用-b 选项的情况下才有效；仅有 MS-DOS 和 MS-Windows 支持该选项
-Z, --null	在文件名的末尾放上 ASCII 空字符以取代换行符

4.13 带选项的 grep(UNIX 和 GNU)

grep 有许多可以控制其行为的选项。并非所有版本的 UNIX 都支持完全相同的选项，所以最好检查帮助手册以得到一个完整的列表。

本节所有的范例均使用下面的 datafile，为方便查阅，它将周期地重复出现。

```
% cat datafile
northwest    NW    Charles Main         3.0    .98    3    34
western      WE    Sharon Gray          5.3    .97    5    23
southwest    SW    Lewis Dalsass        2.7    .8     2    18
southern     SO    Suan Chin            5.1    .95    4    15
southeast    SE    Patricia Hemenway    4.0    .7     4    17
eastern      EA    TB Savage            4.4    .84    5    20
northeast    NE    AM Main Jr.          5.1    .94    3    13
north        NO    Margot Weber         4.5    .89    5    9
central      CT    Ann Stephens         5.7    .94    5    13
```

范例 4-53

```
% grep -n '^south' datafile
3:southwest      SW    Lewis Dalsass        2.7   .8    2   18
4:southern       SO    Suan Chin            5.1   .95   4   15
5:southeast      SE    Patricia Hemenway    4.0   .7    4   17
```

说明

-n 选项在匹配模式成功的行前面加上该行的行号。



范例 4-54

```
% grep -i 'pat' datafile
southeast           SE     Patricia Hemenway     4.0  .7   4  17
```

说明

-i 选项关闭大小写敏感性。表达式 pat 中包含任意的大小写组合都没有关系。

范例 4-55

```
% grep -v 'Suan Chin' datafile
northwest          NW        Charles Main          3.0   .98   3  34
western            WE        Sharon Gray           5.3   .97   5  23
southwest          SW        Lewis Dalsass         2.7   .8    2  18
southeast          SE        Patricia Hemenway     4.0   .7    4  17
eastern            EA        TB Savage             4.4   .84   5  20
northeast          NE        AM Main Jr.           5.1   .94   3  13
north              NO        Margot Weber          4.5   .89   5  9
central            CT        Ann Stephens          5.7   .94   5  13
```

说明

在这里，-v 选项打印所有不包含模式 Suan Chin 的行。这个选项用于从输入文件中删除指定的一个入口。要实际删除此入口，应该将 grep 的输出重定向到一个临时文件，然后将临时文件的名字改回最初文件的名字，如下所示：

```
grep -v 'Suan Chin' datafile > temp
mv temp datafile
```

记得在将 datafile 的输出重定向时一定要使用临时文件。如果直接由 datafile 重定向到 datafile，shell 将摧毁该 datafile(参见 1.6.6 节中的"重定向")。

范例 4-56

```
% grep -l 'SE'  *
datafile
datebook
```

说明

-l 选项导致 grep 仅打印成功匹配模式的文件名，而不打印原文中的行。

范例 4-57

```
% grep -c 'west' datafile
3
```

说明

-c 选项导致 grep 打印成功匹配模式的行数。这并不是指该模式出现的次数。例如，如果一行中 west 出现了 3 次，则该行仅计数一次。

```
% cat datafile
northwest     NW    Charles Main        3.0    .98    3    34
western       WE    Sharon Gray         5.3    .97    5    23
southwest     SW    Lewis Dalsass       2.7    .8     2    18
southern      SO    Suan Chin           5.1    .95    4    15
southeast     SE    Patricia Hemenway   4.0    .7     4    17
eastern       EA    TB Savage           4.4    .84    5    20
northeast     NE    AM Main Jr.         5.1    .94    3    13
north         NO    Margot Weber        4.5    .89    5    9
central       CT    Ann Stephens        5.7    .94    5    13
```

范例 4-58

```
% grep -w 'north' datafile
north            NO      Margot Weber       4.5   .89   5   9
```

说明

-w 选项导致 grep 查找作为一个词[13]而非词的一部分存在的模式。仅打印包含词 north 的行，而不打印包含 northwest,northeast 等词行。

范例 4-59

```
% echo $LOGNAME
lewis
% grep -i "$LOGNAME" datafile
southwest      SW    Lewis Dalsass      2.7   .8   2   18
```

说明

打印了 shell 环境变量 LOGNAME 的值，它包含用户登录名。如果该变量被双引号引用，它还将被 shell 扩展。假如给该变量赋的值多于一个词，则 shell 进行解释时将屏蔽空白。如果是被单引号引用，则不会进行变量替换，也就是说，将会打印$LOGNAME。

GNU grep 选项实例

除 UNIX grep 提供的选项外，GNU 版本提供了进一步精炼模式搜索输出结果的选项。本节所有的例子均使用下面的 datafile，为方便查阅，它将周期地重复出现。

```
% cat datafile
northwest     NW    Charles Main        3.0    .98    3    34
western       WE    Sharon Gray         5.3    .97    5    23
southwest     SW    Lewis Dalsass       2.7    .8     2    18
southern      SO    Suan Chin           5.1    .95    4    15
southeast     SE    Patricia Hemenway   4.0    .7     4    17
```

[13] 词是一个字母数字字符序列，从行首开始或以空白为前导字符，以空白、标点符号或换行符结束。



(续表)

eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 4-60

```
% grep -V
grep (GNU grep) 2.2

    Copyright (C) 1988, 92, 93, 94, 95, 96, 97 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO warranty;
not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

说明

-v 选项列出了 grep 的版本和版权信息。任何发往 GNU 基金会的 bug 报告都应该包含版本信息。

范例 4-61

```
1   % grep -2 Patricia datafile
    southwest       SW      Lewis Dalsass       2.7    .8     2     18
    southern        SO      Suan Chin           5.1    .95    4     15
    southeast       SE      Patricia Hemenway   4.0    .7     4     17
    eastern         EA      TB Savage           4.4    .84    5     20
    northeast       NE      AM Main Jr.         5.1    .94    3     13
2   % grep -C Patricia datafile
    southwest       SW      Lewis Dalsass       2.7    .8     2     18
    southern        SO      Suan Chin           5.1    .95    4     15
    southeast       SE      Patricia Hemenway   4.0    .7     4     17
    eastern         EA      TB Savage           4.4    .84    5     20
    northeast       NE      AM Main Jr.         5.1    .94    3     13
```

说明

1. 在查找到匹配 Patricia 的行后，grep 显示该行及该行的前两行和后两行。
2. -C 选项与-2 相同。

% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 4-62

```
% grep -A 2 Patricia datafile
southeast        SE     Patricia Hemenway     4.0    .7     4     17
eastern          EA     TB Savage             4.4    .84    5     20
northeast        NE     AM Main Jr.           5.1    .94    3     13
```

说明

在查找到匹配 Patricia 的行后，grep 显示该行及该行的后两行。

范例 4-63

```
% grep -B 2 Patricia datafile
southwest        SW     Lewis Dalsass       2.7    .8     2     18
southern         SO     Suan Chin           5.1    .95    4     15
southeast        SE     Patricia Hemenway   4.0    .7     4     17
```

说明

在查找到匹配 Patricia 的行后，grep 显示该行及该行的前两行。

范例 4-64

```
% grep -b '[abc]' datafile
0:northwest       NW     Charles Main        3.0    .98    3    34
39:western        WE     Sharon Gray         5.3    .97    5    23
76:southwest      SW     Lewis Dalsass       2.7    .8     2    18
115:southern      SO     Suan Chin           5.1    .95    4    15
150:southeast     SE     Patricia Hemenway   4.0    .7     4    17
193:eastern       EA     TB Savage           4.4    .84    5    20
228:northeast     NE     AM Main Jr.         5.1    .94    3    13
266:north         NO     Margot Weber        4.5    .89    5     9
301:central       CT     Ann Stephens        5.7    .94    5    13
```

说明

使用-b 选项，grep 在输出的每行前打印该行在输入文件中的偏移字节数

下面的两个范例不再使用 datafile，而是使用文件 negative 以示范-e 和-x 选项的用法。

```
% cat negative
-40 is cold.
This is line 1.
This is line 2.5
-alF are options to the ls command
```

范例 4-65

```
1   % grep -e '-alF' negative
    -alF are options to the ls command
2   % grep --regexp=-40 negative
    -40 is cold.
```



说明

1. 使用-e 选项，grep 将模式中所有的字符同等对待，因此第一个长划线将不会被误当作选项。

2. 表示-e 的另一种方式是– –regexp=pattern，其中 pattern 是正则表达式；在本例中，正则表达式为-40。

范例 4-66

```
% grep -x -e '-40 is cold.' negative
-40 is cold.
```

说明

使用-x 选项，除非搜索模式与整行内容完全一致，否则 grep 不会匹配该行。使用-e 以将一个长划线作为搜索字符串的第一个字符。

本节余下的范例使用下面的 datafile 文件。

```
% cat datafile
northwest    NW    Charles Main         3.0    .98    3    34
western      WE    Sharon Gray          5.3    .97    5    23
southwest    SW    Lewis Dalsass        2.7    .8     2    18
southern     SO    Suan Chin            5.1    .95    4    15
southeast    SE    Patricia Hemenway    4.0    .7     4    17
eastern      EA    TB Savage            4.4    .84    5    20
northeast    NE    AM Main Jr.          5.1    .94    3    13
north        NO    Margot Weber         4.5    .89    5    9
central      CT    Ann Stephens         5.7    .94    5    13
```

范例 4-67

```
1  % cat repatterns
   western
   north
2  % grep -f repatterns datafile
   northwest    NW    Charles Main    3.0    .98    3    34
   western      WE    Sharon Gray     5.3    .97    5    23
   northeast    NE    AM Main Jr.     5.1    .94    3    13
   north        NO    Margot Weber    4.5    .89    5    9
```

说明

1. 显示文件 repatterns。它包含了 grep 的搜索模式，这些搜索模式将与输入文件中的行进行匹配。western 和 north 是 grep 使用的搜索模式。

2. 在-f 选项后跟上一个文件名(本例中为 repatterns)，grep 将从该文件获取搜索模式并与 datafile 中的行进行匹配。grep 搜索并打印所有包含模式 western 和 north 的行。

范例 4-68

```
1  % grep '[0-9]' datafile db
   datafile:northwest NW     Charles Main         3.0   .98   3   34
   datafile:western WE       Sharon Gray          5.3   .97   5   23
   datafile:southwest SW     Lewis Dalsass        2.7   .8    2   18
   datafile:southern SO      Suan Chin            5.1   .95   4   15
   datafile:southeast SE     Patricia Hemenway    4.0   .7    4   17
   datafile:eastern EA       TB Savage            4.4   .84   5   20
   datafile:northeast NE     AM Main Jr.          5.1   .94   3   13
   datafile:north NO         Margot Weber         4.5   .89   5   9
   datafile:central CT       Ann Stephens         5.7   .94   5   13
   db:123
2  % grep -h '[0-9]' datafle db
   northwest       NW       Charles Main         3.0   .98   3   34
   western         WE       Sharon Gray          5.3   .97   5   23
   southwest       SW       Lewis Dalsass        2.7   .8    2   18
   southern        SO       Suan Chin            5.1   .95   4   15
   southeast       SE       Patricia Hemenway    4.0   .7    4   17
   eastern         EA       TB Savage            4.4   .84   5   20
   northeast       NE       AM Main Jr.          5.1   .94   3   13
   north           NO       Margot Weber         4.5   .89   5   9
   central         CT       Ann Stephens         5.7   .94   5   13
   123
```

说明

1. 如果列出的文件多于一个，grep 在输出每行内容之前先输出文件名。文件名为 datafile 和 db。

2. 使用-h 选项，grep 禁止头部信息。也就是说，不打印文件名。

范例 4-69

```
% grep -q Charles datafile        或
% grep --quiet Charles datafile
% echo $status
0
```

说明

quiet 选项禁止从 grep 输出。它用于退出状态恰为所需要的值时。当退出状态为 0 时，说明 grep 查找到匹配的模式。

习题 1：grep 练习

(参考从本书合作站点下载的文件中名为 databook 的文件)。

Steve Blenheim:238-923-7366:95 Latham Lane, Easton, PA 83755:11/12/56:20300

Betty Boop:245-836-8357:635 Cutesy Lane, Hollywood, CA 91464:6/23/23:14500

Igor Chevsky:385-375-8395:3567 Populus Place, Caldwell, NJ 23875:6/18/68:23400

Norma Corder:397-857-2735:74 Pine Street, Dearborn, MI 23874:3/28/45:245700

Jennifer Cowan:548-834-2348:583 Laurel Ave., Kingsville, TX 83745:10/1/35:58900



Jon DeLoach:408-253-3122:123 Park St., San Jose, CA 04086:7/25/53:85100
Karen Evich:284-758-2857:23 Edgecliff Place, Lincoln, NB 92743:7/25/53:85100
Karen Evich:284-758-2867:23 Edgecliff Place, Lincoln, NB 92743:11/3/35:58200
Karen Evich:284-758-2867:23 Edgecliff Place, Lincoln, NB 92743:11/3/35:58200
Fred Fardbarkle:674-843-1385:20 Parak Lane, Duluth, MN 23850:4/12/23:780900
Fred Fardbarkle:674-843-1385:20 Parak Lane, Duluth, MN 23850:4/12/23:780900
Lori Gortz:327-832-5728:3465 Mirlo Street, Peabody, MA 34756:10/2/65:35200
Paco Gutierrez:835-365-1284:454 Easy Street, Decatur, IL 75732:2/28/53:123500
Ephram Hardy:293-259-5395:235 CarltonLane, Joliet, IL 73858:8/12/20:56700
James Ikeda:834-938-8376:23445 Aster Ave., Allentown, NJ 83745:12/1/38:45000
Barbara Kertz:385-573-8326:832 Ponce Drive, Gzary, IN 83756:12/1/46:268500
Lesley Kirstin:408-456-1234:4 Harvard Square, Boston, MA 02133:4/22/62:52600
William Kopf:846-836-2837:6937 Ware Road, Milton, PA 93756:9/21/46:43500
Sir Lancelot:837-835-8257:474 Camelot Boulevard, Bath, WY 28356:5/13/69:24500
Jesse Neal:408-233-8971:45 Rose Terrace, San Francisco, CA 92303:2/3/36:25000
Zippy Pinhead:834-823-8319:2356 Bizarro Ave., Farmount, IL 84357:1/1/67:89500
Arthur Putie:923-835-8745:23 Wimp Lane, Kensington, DL 38758:8/31/69:126000
Popeye Sailor:156-454-3322:945 Bluto Street, Anywhere, USA 29358:3/19/35:22350
Jose Santiago:385-898-8357:38 Fife Way, Abilene, TX 39673:1/5/58:95600
Tommy Savage:408-724-0140:1222 Oxbow Court, Sunnyvale, CA 94087:5/19/66:34200
Yukio Takeshida:387-827-1095:13 Uno Lane, Ashville, NC 23556:7/1/29:57000
Vinh Tranh:438-910-7449:8235 Maple Street, Wilmington, VM 29085:9/23/63:68900

1. 打印所有包含字符串 San 的行。
2. 打印所有名字以 J 开头的行。
3. 打印所有以 700 结尾的行。
4. 打印所有不包含 834 的行。
5. 打印所有生日在 December 的行。
6. 打印所有电话号码区号为 408 的行。
7. 打印所有包含一个大写字母，后跟 4 个小写字母，一个逗号，一个空格和一个大写字母的行。
8. 打印所有最后一个名字以 K 或 k 开始的行。
9. 打印所有薪水为 6 位数字的行，前导是一个行号。
10. 打印包含 Lincoln 或 lincoln(注意，grep 不区分大小写)的行。



chapter 5 流编辑器 sed



5.1 sed 简介

sed 是一种新型的，非交互式的编辑器。它能执行与编辑器 vi 和 ex 相同的编辑任务。sed 编辑器没有提供交互使用方式，使用者只能在命令行输入编辑命令、指定文件名，然后在屏幕上查看输出。sed 编辑器没有破坏性，它不会修改文件，除非使用 shell 重定向来保存输出结果。默认情况下，所有的输出行都被打印到屏幕上。

sed 编辑器在 shell 脚本中很有用，因为在 shell 脚本中使用像 vi 或 ex 这类交互式编辑器，要求脚本用户精通该编辑器，而且还会导致用户对打开的文件做出不需要的修改。如果需要执行多项编辑任务，或是不想为给 shell 命令行上的 sed 命令加引号[①]而操心，也可以把 sed 命令写在一个叫做 sed 脚本的文件里。

5.2 sed 的不同版本

Linux 使用的 sed 是 GNU 版的，版权归自由软件基金会所有。GNU 版 grep 与标准 UNIX 发行版本提供的 sed 几乎完全相同。

范例 5-1

```
% sed -V  or  sed --version
GNU sed version 3.02
Copyright (C) 1998 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE, to the extent permitted by law.
```

---

① 记住，在命令行键入命令时，shell 会设法对任何元字符和空格进行转换，sed 命令中的任何可能被转换的字符都必须使用引号括起来。



5.3 sed 的工作过程

sed 编辑器逐行处理文件(或输入)，并将输出结果发送到屏幕。sed 的命令就是在 vi 和 ed/ex 编辑器中见到的那些。sed 把当前正在处理的行保存在一个临时缓冲区中，这个缓冲区称为模式空间或临时缓冲。sed 处理完模式空间中的行后(即在该行上执行完 sed 命令后)，就把该行发送到屏幕上(除非之前有命令删除这一行或取消打印操作)。sed 每处理完一行就将其从模式空间中删除，然后将下一行读入空间，进行处理和显示。处理完输入文件的最后一行后，sed 便结束运行。sed 把每一行都存在临时缓冲区中，对这个副本进行编辑，所以不会修改或破坏原文件。



5.4 正则表达式

与 grep 一样，sed 在文件中查找模式时也要使用正则表达式(RE)和各种元字符(参见 5.8 节中的表 5-3)。正则表达式是括在斜杠间的模式，用于查找和替换。

```
sed -n '/RE/p' filename
sed -n '/love/p' filename

sed -n 's/RE/replacement string/' filename
sed -n 's/love/like/' filename
```

如果要把正则表达式分隔符改成另一个字符，比如 c，只要在这个字符前加一个反斜杠，在字符后跟上正则表达式，再跟上这个字符即可。请看下面这个示例。

```
sed -n '/love/p' filename
```

这条命令会打印出文件 filename 中所有包含 love 的行。下面这条命令将更换分隔符。

```
sed -n '\cREcp' filename
```

字母 c 顶替斜杠，成为分隔正则表达式的字符。

范例 5-2

```
1   % sed -n '/12\/10\/04/p' datafile
2   % sed -n '\x12/10/04xp' datafile       # sed lets you change the delimiter
```

说明

1. 如果斜杠本身是正则表达式的一部分，必须在它前面加上反斜杠，以免和用作分隔

符的斜杠混淆。

2. x 顶替斜杠成为分隔符，便于在正则表达式中包含斜杠。

前面曾经提到过，如果在文件中找到指定的模式，grep 将返回退出状态 0；如果没找到，就返回 1。sed 不一样，不管是否找到指定模式，它的退出状态都是 0。只有当命令存在语法错误时，sed 的退出状态才不是 0(参见 5.7 节“报错信息和退出状态”)。下面这个示例中，grep 和 sed 命令分别用来在文件中查找正则表达式 John。

正则表达式还可以被用作地址的一部分，下一节“定址”会对此进行介绍。

范例 5-3

```
1   % grep 'John' datafile        # grep searches for John
2   % echo $status
    1

3   % sed -n '/John/p' datafile   # sed searches for John
4   % echo $status
    0
```

说明

1. 使用 grep 时，正则表达式 John 没有包含在分隔符中。
2. 如果找到了模式 John，grep 命令的退出状态为 0，否则就不为 0。
3. sed 将打印出所有包含正则表达式模式 John 的行。
4. 即便没有在文件中找到模式 John，退出状态也是 0，因为命令的语法是正确的。

5.5 定址

定址用于决定对哪些行进行编辑。地址的形式可以是数字、正则表达式(又称为“上下文地址”)或二者的结合。如果没有指定地址，sed 将处理输入文件中的所有行。

如果指定的地址是一个数字，则这个数字代表行号。美元符号可用来指代输入文件的最后一行。如果给出的是逗号分隔的两个行号，那么需要处理的地址就是这两行之间的范围(包括这两行在内)。范围可以是数字、正则表达式或二者的组合。

sed 命令告诉 sed 对指定行进行何种操作，包括打印、删除、修改等。

格式

sed 'command' filename(s)

范例 5-4

```
1   sed '1,3d' myfile
2   sed -n '/[Jj]ohn/p' datafile
```

说明

1. 删除文件 myfile 的第 1~3 行。
2. 只打印文件中与模式 John 或 john 匹配的行。



5.6 命令与选项

sed 命令告诉 sed 如何处理由地址指定的各输入行。如果没有指定地址，sed 就会处理输入的所有行(后面示例中%是 csh 的提示符)。请参考表 5-1 中列出的 sed 命令及其功能，表 5-2 则列出了命令选项并说明了它们如何控制 sed 的行为。

表 5-1 sed 命令

命令	功能
a\	在当前行后添加一行或多行
c\	用新文本修改(替换)当前行中的文本
d	删除行
i\	在当前行之前插入文本
h	把模式空间里的内容复制到暂存缓冲区
H	把模式空间里的内容追加到暂存缓冲区
g	取出暂存缓冲区的内容，将其复制到模式空间，覆盖该处原有内容
G	取出暂存缓冲区的内容，将其复制到模式空间，追加在原有内容后面
l	列出非打印字符
p	打印行
n	读入下一输入行，并从下一条命令而不是第一条命令开始对其的处理
q	结束或退出 sed
r	从文件中读取输入行
!	对所选行以外的所有行应用命令
s	用一个字符串替换另一个
替换标志	
g	在行内进行全局替换
p	打印行
w	将行写入文件
x	交换暂存缓冲区与模式空间的内容
y	将字符转换为另一字符(不能对正则表达式使用 y 命令)

表 5-2 sed 选项

选项	功能
-e	允许多项编辑
-f	指定 sed 脚本文件名
-n	取消默认的输出

如果需要使用多条命令，或者需要在某个地址范围内嵌套地址，就必须用花括号将命令括起来，每行只写一条命令，或者用分号分隔同一行中的多条命令。

感叹号(！)用于否定的命令。例如：

```
sed '/Tom/d' file
```

这条命令告诉 sed 删除所有包含模式 Tom 的行，而命令

```
sed '/Tom/!d' file  (sh, ksh, bash)
sed '/Tom/\!d' file  (csh, tcsh)
```

则让 sed 删除所有不含 Tom 的行。

sed 的选项包括–e、–f 和–n。-e 用于在命令行上指定多项编辑；–f 后面跟 sed 脚本文件名；-n 则用于取消打印输出。

5.6.1 用 sed 修改文件

sed 是不具破坏性的编辑器。sed 在屏幕上显示编辑的结果，但不会改变所编辑的文件。如果要将编辑结果真正反映在文件中，就必须把输出重定向到另一个文件，然后重命名原文件。

范例 5-5

```
1  % sed '1,3d' filex >  temp
2  % mv temp filex
```

说明

1. 删除文件 filex 的 1~3 行。把剩下的行重定向到文件 temp，而不显示在屏幕上(请确保发送输出结果的目标文件(本例中的文件 temp)是一个空文件。否则，重定向会破坏该文件的内容)。
2. mv 命令将 temp 的内容覆盖到 filex。

5.6.2 GNU sed 的选项

范例 5-6 列出了 GNU sed 提供的更多选项，以及这些选项如何控制 sed 的行为。如果指定了选项-h，sed 将显示它的命令行选项列表，且每个选项后面都有它的功能简介。

范例 5-6

```
% sed -h
Usage: sed [OPTION]... {script-only-if-no-other-script} [input-file]...

   -n, --quiet, --silent
                 suppress automatic printing of pattern space
   -e script, --expression=script
                 add the script to the commands to be executed
   -f script-file, --file=script-file
                 add the contents of script-file to the commands to be executed
   --help         display this help and exit
   -V, --version output version information and exit
```



说明

如果没有指定-e、--expression、-f 或--file 选项，sed 就将第一个非选项的命令行变量作为将要解析执行的 sed 脚本，其余的命令行变量被视为输入文件名。如果没有指定输入文件，sed 将从标准输入读取输入。

5.7 报错信息和退出状态

遇到语法错误时，sed 会向标准错误输出发送一条相当简单的报错信息。但是，如果 sed 判断不出错在何处，它会“断章取义”，给出令人迷惑的报错信息。如果没有语法错误，sed 就会返回给 shell 一个退出状态，状态为 0 代表成功，为非 0 整数则代表失败。[2]

范例 5-7

```
1  % sed '1,3v ' file
   sed: Unrecognized command: 1,3v
   % echo $status           # use echo $? if using Korn or Bourne shell
   2

2  % sed '/^John' file
   sed: Illegal or missing delimiter: /^John

3  % sed 's/134345/g' file
   sed: Ending delimiter missing on substitution: s/134345/g
```

说明

1. sed 不能识别命令 v，因此退出状态为 2，表示 sed 因语法问题而退出。
2. 模式/^John 缺少了结尾的斜杠。
3. 替换命令 s 包含了查找串却遗漏了替换串。

5.8 元字符

与 grep 一样，sed 也支持很多特殊的元字符，用它们来控制模式查找。参见表 5-3。

表 5-3 sed 的正则表达式元字符

元字符	功 能	示 例	示例的匹配对象
^	行首定位符	/^love/	匹配所有以 love 开头的行
$	行尾定位符	/love$/	匹配所有以 love 结尾的行
.	匹配除换行符外的单个字符	/l..e/	匹配包含字母 l、后跟两个任意字符、再跟字母 e 的行

② 如果要了解完整的诊断信息，请参考 UNIX 的 sed 手册。

(续表)

元字符	功　能	示　例	示例的匹配对象
*	匹配零个或多个前导字符	/ *love/	匹配在零个或多个空格紧跟着模式 love 的行
[ ]	匹配指定字符组内的任一字符	/[Ll]ove/	匹配包含 love 和 Love 的行
[^]	匹配不在指定字符组内的任一字符	/[^A-KM-Z]ove/	匹配包含 ove，但 ove 之前的那个字符不在 A 至 K 或 M 至 Z 之间的行
\(..\)	保存已匹配的字符	s/\(love\)able/\1er/	标记元字符之间的模式，并将其保存为标签 1，之后可以用\1 来引用它。最多可定义 9 个标签，从左边开始编号，最左边的是第一个。在这个示例中，love 被保存在寄存器 1 里，之后被替换串引用，结果 loveable 被替换为 lover。
&	保存查找串以便在替换串中引用	s/love/**&**/	符号&代表查找串。字符串 love 将替换前后各加了两个星号的引用，即 love 变成**love**
\<	词首定位符	/\<love/	匹配包含以 love 开头的单词的行
\>	词尾定位符	/love\>/	匹配包含以 love 结尾的单词的行
x\{m\}	连续 m 个 x	/o\{5\}/	分别匹配出现连续 5 个字母 o、至少 5 个连续的 o、或 5~10 个连续的 o 的行
x\{m,\}	至少 m 个 x	/o\{5,\}/	
x\{m,n\}[③]	至少 m 个、但不超过 n 个 x	/o\{5,10\}/	

5.9 sed 范例

下面这组范例展示了如何使用 sed，包括如何使用它的选项、命令和正则表达式。请记住 sed 是非破坏性的，它不会修改正在编辑的文件，除非像范例 5-6 那样重定向它的输出结果。

本节中的范例依旧使用 datafile 作为输入文件。为了方便读者阅读，再一次给出文件的内容如下所示。

③ 并不是所有版本的 UNIX 和模式匹配工具都支持元字符\{和\}，不过，vi 和 grep 通常都支持它们。



% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

5.9.1 打印：p 命令

命令 p 是打印命令，用于显示模式缓冲区的内容。默认情况下，sed 把输入行打印在屏幕上，选项-n 用于取消默认的打印操作。当选项-n 和命令 p 同时出现时，sed 可打印选定的内容。

范例 5-8

```
% sed '/north/p' datafile
northwest          NW     Charles Main        3.0   .98    3    34
northwest          NW     Charles Main        3.0   .98    3    34
western            WE     Sharon Gray         5.3   .97    5    23
southwest          SW     Lewis Dalsass       2.7   .8     2    18
southern           SO     Suan Chin           5.1   .95    4    15
southeast          SE     Patricia Hemenway   4.0   .7     4    17
eastern            EA     TB Savage           4.4   .84    5    20
northeast          NE     AM Main Jr.         5.1   .94    3    13
northeast          NE     AM Main Jr.         5.1   .94    3    13
north              NO     Margot Weber        4.5   .89    5     9
north              NO     Margot Weber        4.5   .89    5     9
central            CT     Ann Stephens        5.7   .94    5    13
```

说明

默认情况下，sed 把所有输入行都打印在标准输出上。如果在某一行匹配到模式 north，sed 将把该行另外打印一遍。

范例 5-9

```
% sed -n '/north/p' datafile
northwest          NW     Charles Main       3.0   .98    3    34
northeast          NE     AM Main Jr.        5.1   .94    3    13
north              NO     Margot Weber       4.5   .89    5     9
```

说明

默认情况下，sed 打印当前模式缓冲区中的输入行。命令 p 指示 sed 将再次打印该行。选项-n 取消 sed 的默认打印动作。选项-n 与命令 p 配合使用时，模式缓冲区内的输入行只被打印一次。如果不指定-n 选项，sed 就会像上个示例中那样打印出重复的输出行。如果指定了选项-n，则 sed 只打印包含模式 north 的那几行。

5.9.2 删除：d 命令

命令 d 用于删除输入行。sed 先将输入行从文件复制到模式缓冲区，然后对该行执行 sed 命令，最后将模式缓冲区的内容显示在屏幕上。如果发出的是命令 d，当前模式缓冲区里的输入行会被删除，不被显示。

```
% cat datafile
northwest    NW    Charles Main         3.0    .98    3    34
western      WE    Sharon Gray          5.3    .97    5    23
southwest    SW    Lewis Dalsass        2.7    .8     2    18
southern     SO    Suan Chin            5.1    .95    4    15
southeast    SE    Patricia Hemenway    4.0    .7     4    17
eastern      EA    TB Savage            4.4    .84    5    20
northeast    NE    AM Main Jr.          5.1    .94    3    13
north        NO    Margot Weber         4.5    .89    5    9
central      CT    Ann Stephens         5.7    .94    5    13
```

范例 5-10

```
% sed '3d' datafile
northwest    NW    Charles Main         3.0    .98    3    34
western      WE    Sharon Gray          5.3    .97    5    23
southern     SO    Suan Chin            5.1    .95    4    15
southeast    SE    Patricia Hemenway    4.0    .7     4    17
eastern      EA    TB Savage            4.4    .84    5    20
northeast    NE    AM Main Jr.          5.1    .94    3    13
north        NO    Margot Weber         4.5    .89    5     9
central      CT    Ann Stephens         5.7    .94    5    13
```

说明

删除第 3 行。默认情况下，其余的行都被打印到屏幕上。

范例 5-11

```
% sed '3,$d' datafile
northwest    NW    Charles Main         3.0    .98    3    34
western      WE    Sharon Gray          5.3    .97    5    23
```

说明

删除从第 3 行到最后一行的内容。美元符($)代表文件的最后一行。逗号被称为范围操



作符。剩余各行被打印。本例中，地址范围开始于第3行，结束于美元符($)代表的最后一行。

范例 5-12

```
% sed '$d' datafile
northwest      NW     Charles Main        3.0   .98   3   34
western        WE     Sharon Gray         5.3   .97   5   23
southwest      SW     Lewis Dalsass       2.7   .8    2   18
southern       SO     Suan Chin           5.1   .95   4   15
southeast      SE     Patricia Hemenway   4.0   .7    4   17
eastern        EA     TB Savage           4.4   .84   5   20
northeast      NE     AM Main Jr.         5.1   .94   3   13
north          NO     Margot Weber        4.5   .89   5    9
```

说明

删除最后一行。美元符($)代表文件的最后一行。默认情况下，将打印所有未受 d 命令影响的行。

范例 5-13

```
% sed '/north/d' datafile
western        WE     Sharon Gray         5.3   .97   5   23
southwest      SW     Lewis Dalsass       2.7   .8    2   18
southern       SO     Suan Chin           5.1   .95   4   15
southeast      SE     Patricia Hemenway   4.0   .7    4   17
eastern        EA     TB Savage           4.4   .84   5   20
central        CT     Ann Stephens        5.7   .94   5   13
```

说明

所有包含模式 north 的行都被删除；其余的行被打印。

5.9.3 替换：s 命令

命令 s 是替换命令。替换和取代文件中的文本可以通过 sed 中的 s 命令来实现，s 后包含在斜杠中的文本是正则表达式，后面跟着的是需要替换成的文本。可以通过 g 标志对行进行全局替换。

% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 5-14

```
% sed 's/west/north/g' datafile
northnorth      NW      Charles Main        3.0   .98   3   34
northern        WE      Sharon Gray         5.3   .97   5   23
southnorth      SW      Lewis Dalsass       2.7   .8    2   18
southern        SO      Suan Chin           5.1   .95   4   15
southeast       SE      Patricia Hemenway   4.0   .7    4   17
eastern         EA      TB Savage           4.4   .84   5   20
northeast       NE      AM Main Jr.         5.1   .94   3   13
north           NO      Margot Weber        4.5   .89   5   9
central         CT      Ann Stephens        5.7   .94   5   13
```

说明

s 命令用于替换。命令末端的标志 g 表示在行内进行全局替换；也就是说，如果某一行里出现了多个 west，所有的 west 都被替换为 north。如果没有 g 命令，则只将每一行的第一个 west 替换为 north。

范例 5-15

```
% sed -n 's/^west/north/p' datafile
northern        WE      Sharon Gray         5.3   .97   5   23
```

说明

s 命令用于替换。选项-n 与命令末尾的标志 p 配合，告诉 sed 只打印发生替换的那些行；也就是说，如果只有在行首找到 west 并将其替换为 north 时，才打印此行。

范例 5-16

```
% sed 's/[0-9][0-9]$/&.5/' datafile
northwest       NW      Charles Main        3.0   .98   3   34.5
western         WE      Sharon Gray         5.3   .97   5   23.5
southwest       SW      Lewis Dalsass       2.7   .8    2   18.5
southern        SO      Suan Chin           5.1   .95   4   15.5
southeast       SE      Patricia Hemenway   4.0   .7    4   17.5
eastern         EA      TB Savage           4.4   .84   5   20.5
northeast       NE      AM Main Jr.         5.1   .94   3   13.5
north           NO      Margot Weber        4.5   .89   5   9
central         CT      Ann Stephens        5.7   .94   5   13.5
```

说明

当“与”符号[4](&)用在替换串中时，它代表在查找串中匹配到的内容。这个示例中，所有以两位数结尾的行后面都被加上.5。

范例 5-17

```
% sed -n 's/Hemenway/Jones/gp' datafile
southeast       SE      Patricia Jones      4.0   .7    4   17
```

④ 如果要在替换串中表示“与”号的字面含义，就要对其进行转义，记为：\&。



说明

文件中出现的所有 Hemenway 都被替换为 Jones，只有发生变化的行才被打印。选项-n 与命令 p 的组合取消了默认的输出。标志 g 的含义是在行内进行全局替换。

范例 5-18

```
% sed -n 's/\(Mar\)got/\1ianne/p' datafile
north           NO      Marianne Weber      4.5   .89   5   9
```

说明

包含在圆括号里的模式 Mar 作为标签 1 保存于特定的寄存器中。替换串可通过\1 引用它。则 Margot 被替换为 Marianne。

% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 5-19

```
% sed 's#3#88#g' datafile
northwest       NW      Charles Main        88.0  .98   88  884
western         WE      Sharon Gray         5.88  .97   5   288
southwest       SW      Lewis Dalsass       2.7   .8    2   18
southern        SO      Suan Chin           5.1   .95   4   15
southeast       SE      Patricia Hemenway   4.0   .7    4   17
eastern         EA      TB Savage           4.4   .84   5   20
northeast       NE      AM Main Jr.         5.1   .94   88  188
north           NO      Margot Weber        4.5   .89   5   9
central         CT      Ann Stephens        5.7   .94   5   188
```

说明

紧跟在 s 命令后的字符就是查找串和替换串之间的分隔符。分隔符默认为正斜杠，但可以改变。无论什么字符(换行符、反斜线除外)，只要紧跟着 s 命令，就成了新的串分隔符。这个方法在查找包含正斜杠的模式时很管用，例如查找路径名或生日。

5.9.4 指定行的范围：逗号

行的范围从文件中一个地址开始，在另一个地址结束。地址范围可以是行号(例如 5，10)，正则表达式(例如/Dick/和/Joe/)，或者两者的结合(例如/north/,$)。范围是闭合的——包

含开始条件的行，结束条件的行，以及两者之间的行。如果结束条件无法满足，就会一直操作到文件结尾。如果结束条件满足，则继续查找满足开始条件的位置，范围重新开始。

范例 5-20

```
% sed -n '/west/,/east/p' datafile
northwest   NW   Charles Main       3.0   .98   3   34
western     WE   Sharon Gray        5.3   .97   5   23
southest    SW   Lewis Dalsass      2.7   .8    2   18
southern    SO   Suan Chin          5.1   .95   4   15
southeast   SE   Patricia Hemenway  4.0   .7    4   17
```

说明

打印在模式 west 和 east 之间的所有行。如果 west 出现在 east 之后的某一行，则打印的范围从 west 所在行开始，到下一个出现 east 的行或文件末尾(如果前者未出现)。图中用箭头标出了该范围。

范例 5-21

```
% sed -n '5,/^northeast/p' datafile
southeast          SE      Patricia Hemenway      4.0   .7    4   17
eastern            EA      TB Savage              4.4   .84   5   20
northeast          NE      AM Main Jr.            5.1   .94   3   13
```

说明

打印从第 5 行到第一个以 northeast 开头的行之间的所有行。

范例 5-22

```
% sed '/west/,/east/s/$/**VACA**/' datafile
northwest   NW   Charts Main        3.0   .98   3   34**VACA**
western     WE   Sharon Gray        5.3   .97   5   23**VACA**
southest    SW   Lewis Dalsass      2.7   .8    2   18**VACA**
southern    SO   Suan Chin          5.1   .95   4   15**VACA**
southeast   SE   Patricia Hemenway  4.0   .7    4   17**VACA**
eastern     EA   TB Savage          4.4   .84   5   20
northeast   NE   AM Main Jr.        5.1   .94   3   13
north       NO   Margot Weber       4.5   .89   5   9
central     CT   Ann Stephens       5.7   .94   5   13
```

说明

修改从模式 east 和 west 之间的所有行，将各行的行尾($)替换为字符串**VACA**。换行符被移到这个新的字符串后面。箭头标出了范围。

5.9.5 多重编辑：e 命令

-e 命令是编辑命令，用于 sed 执行多个编辑任务的情况下。在下一行开始编辑前，所有的编辑动作将应用到模式缓冲区中的行上。



% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 5-23

```
% sed -e '1,3d' -e 's/Hemenway/Jones/' datafile
southern         SO    Suan Chin          5.1   .95   4   15
southeast        SE    Patricia Jones     4.0   .7    4   17
eastern          EA    TB Savage          4.4   .84   5   20
northeast        NE    AM Main Jr.        5.1   .94   3   13
north            NO    Margot Weber       4.5   .89   5   9
central          CT    Ann Stephens       5.7   .94   5   13
```

说明

选项-e 用于进行多重编辑。第一重编辑删除第 1~3 行。第二重编辑将 Hemenway 替换为 Jones。因为是逐行进行这两项编辑(即这两个命令都在模式空间的当前行上执行)，所以编辑命令的顺序会影响结果。例如，如果两条命令执行的都是替换，则前一次替换会影响后一次替换。

5.9.6 读文件：r 命令

r 命令是读命令。sed 使用该命令将一个文本文件中的内容加到当前文件的特定位置上。

范例 5-24

```
% cat newfile
                     ------------------------------------
                     |  ***SUAN HAS LEFT THE COMPANY***  |
                     ------------------------------------
% sed '/Suan/r newfile' datafile
northwest        NW     Charles Main       3.0   .98   3   34
western          WE     Sharon Gray        5.3   .97   5   23
southwest        SW     Lewis Dalsass      2.7   .8    2   18
southern         SO     Suan Chin          5.1   .95   4   15
                     -----------------------------------
                     | ***SUAN HAS LEFT THE COMPANY***  |
                     |__________________________________|
southeast        SE     Patricia Hemenway  4.0   .7    4   17
eastern          EA     TB Savage          4.4   .84   5   20
```

```
northeast          NE      AM Main Jr.          5.1   .94   3    13
north              NO      Margot Weber         4.5   .89   5    9
central            CT      Ann Stephens         5.7   .94   5    13
```

说明

r 命令读取文件的指定行。如果在文件 datafile 的某一行匹配到模式 Suan，就在该行后读入文件 newfile 的内容。如果出现 Suan 的不止一行，则在出现 Suan 的各行后都读入 newfile 文件的内容。

5.9.7 写文件：w 命令

w 命令是写命令，sed 使用该命令将当前文件中的一些行写到另一个文件中。

范例 5-25

```
% sed -n '/north/w newfile' datafile
cat newfile
northwest          NW      Charles Main         3.0   .98   3    34
northeast          NE      AM Main Jr.          5.1   .94   3    13
north              NO      Margot Weber         4.5   .89   5    9
```

说明

w 命令把指定行写入文件。文件 datafile 中所有包含模式 north 的行都被写到文件 newfile 中。

5.9.8 追加：a 命令

a 命令是追加命令，追加将添加新文本到文件中当前行(即读入模式缓冲区中的行)的后面。不管是在命令行中，还是在 sed 脚本中，a 命令总是在反斜杠的后面。如果在 C/TC shell 命令行中，则需要两个反斜杠，如范例 5-26 所示——一个斜线用于 sed 追加命令，另一个用于行的继续。在 C/TC shell 中，用于 sed 命令的两个引号必须位于同一行。因此，第二个反斜杠使得在下一行继续键入而不会导致引号不匹配错误(Bourne shell 和 Bash shell 不需要第二个反斜杠，因为两个引号不需要在同一行上)。所追加的文本行位于 sed 命令的下方另起一行。如果要追加的内容超过行，则每一行都必须以反斜杠结束，最后一行除外。最后一行将以引号和文件名结束。

% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13



范例 5-26

```
% sed '/^north /a\\      # Use two backslashes only if the shell is C/TC
--->THE NORTH SALES DISTRICT HAS MOVED<---' datafile
northwest        NW    Charles Main       3.0  .98  3  34
western          WE    Sharon Gray        5.3  .97  5  23
southwest        SW    Lewis Dalsass      2.7  .8   2  18
southern         SO    Suan Chin          5.1  .95  4  15
southeast        SE    Patricia Hemenway  4.0  .7   4  17
eastern          EA    TB Savage          4.4  .84  5  20
northeast        NE    AM Main Jr.        5.1  .94  3  13
north            NO    Margot Weber       4.5  .89  5  9
--->THE NORTH SALES DISTRICT HAS MOVED<---
central          CT    Ann Stephens       5.7  .94  5  13
```

说明

命令 a 用于追加。字符串--->THE NORTH SALES DISTRICT HAS MOVED<---被加在以 north 开头，north 后跟一空格的各行之后。用于追加的文本必须出现在追加命令的下一行上。

sed 要求 a 命令后面跟一个反斜杠。本例中的第二个反斜杠被 C shell 用来转义换行符，这样右侧的引号的就能在下一行出现[⑤]。如果要追加的内容超过一行，则除最后一行外，其他各行都必须以反斜杠结尾。

5.9.9 插入：i 命令

i 命令是插入命令，类似于 a 命令，但是不是在当前行后增加文本，而是在当前行的前面插入新的文本，即刚读入模式缓冲区中的行。在命令行或者 sed 脚本中，i 命令后都带一个反斜杠。

范例 5-27

```
% sed '/eastern/i\\      # Use two backslashes for C/TC
NEW ENGLAND REGION\\
-------------------------------------' datafile
northwest        NW    Charles Main       3.0  .98  3  34
western          WE    Sharon Gray        5.3  .97  5  23
southwest        SW    Lewis Dalsass      2.7  .8   2  18
southern         SO    Suan Chin          5.1  .95  4  15
southeast        SE    Patricia Hemenway  4.0  .7   4  17
NEW ENGLAND REGION
-------------------------------------
eastern          EA    TB Savage          4.4  .84  5  20
northeast        NE    AM Main Jr.        5.1  .94  3  13
north            NO    Margot Weber       4.5  .89  5  9
central          CT    Ann Stephens       5.7  .94  5  13
```

说明

命令 i 是插入命令。如果在某一行匹配到模式 eastern，i 命令就在该行的上方插入命令

⑤ Bourne shell 和 Korn shell 都不需要用第二个反斜杠来转义换行符，因为它们只要求引号成对出现，但不一定要在同一行内配对。

中反斜杠后的文本。除了最后一行，插入的文本中每一行都必须以反斜杠结尾(示例中额外的反斜杠专门用于满足 C shell 的要求)。

5.9.10 修改：c 命令

c 命令是修改命令。sed 使用该命令将已有文本修改成新的文本。旧文本被覆盖。

```
% cat datafile
northwest    NW    Charles Main         3.0    .98    3    34
western      WE    Sharon Gray          5.3    .97    5    23
southwest    SW    Lewis Dalsass        2.7    .8     2    18
southern     SO    Suan Chin            5.1    .95    4    15
southeast    SE    Patricia Hemenway    4.0    .7     4    17
eastern      EA    TB Savage            4.4    .84    5    20
northeast    NE    AM Main Jr.          5.1    .94    3    13
north        NO    Margot Weber         4.5    .89    5    9
central      CT    Ann Stephens         5.7    .94    5    13
```

范例 5-28

```
% sed '/eastern/c\\
  THE EASTERN REGION HAS BEEN TEMPORARILY CLOSED' datafile
northwest    NW    Charles Main         3.0    .98    3    34
western      WE    Sharon Gray          5.3    .97    5    23
southwest    SW    Lewis Dalsass        2.7    .8     2    18
southern     SO    Suan Chin            5.1    .95    4    15
southeast    SE    Patricia Hemenway    4.0    .7     4    17
  THE EASTERN REGION HAS BEEN TEMPORARILY CLOSED
northeast    NE    AM Main Jr.          5.1    .94    3    13
north        NO    Margot Weber         4.5    .89    5    9
central      CT    Ann Stephens         5.7    .94    5    13
```

说明

c 命令是修改命令。该命令将完整地修改在模式缓冲区中的当前行。如果模式 eastern 被匹配，c 命令将用反斜杠后的文本替换包含 eastern 的行。除最后一行之外，反斜杠对每一个插入的行都是必要的(附加的反斜杠用于 C shell 的特定要求)。

5.9.11 获取下一行：n 命令

n 命令表示下一条命令。sed 使用该命令获取输入文件的下一行，并将其读入到模式缓冲区中，任何 sed 命令都将应用到匹配行紧接着的下一行上。

范例 5-29

```
% sed '/eastern/{ n; s/AM/Archie/; }' datafile
northwest    NW    Charles Main         3.0    .98    3    34
western      WE    Sharon Gray          5.3    .97    5    23
southwest    SW    Lewis Dalsass        2.7    .8     2    18
southern     SO    Suan Chin            5.1    .95    4    15
```



```
  southeast      SE    Patricia Hemenway   4.0  .7   4  17
  eastern        EA    TB Savage           4.4  .84  5  20
→ northeast      NE    Archie Main Jr.     5.1  .94  3  13
  north          NO    Margot Weber        4.5  .89  5  9
  central        CT    Ann Stephens        5.7  .94  5  13
```

说明

如果在某一行匹配到模式 eastern，n 命令就指示 sed 用下一个输入行(即包含 AM Main Jr 的那行)替换模式空间中的当前行，并用 Archie 替换该行中的 AM，然后打印该行，再继续往下处理。

5.9.12 转换：y 命令

y 命令表示转换。该命令与 UNIX/Linux 中的 tr 命令相似，字符按照一对一的方式从左到右进行转换。例如，命令 y/abc/ABC 将把所有小写的 a 转换成 A，小写的 b 转换成 B，小写的 c 转换成 C。

范例 5-30

```
% sed '1,3y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKL
  MNOPQRSTUVWXYZ/'  datafile
→ NORTHWEST      NW    CHARLES MAIN        3.0  .98  3  34
  WESTERN        WE    SHARON GRAY         5.3  .97  5  23
→ SOUTHWEST      SW    LEWIS DALSASS       2.7  .8   2  18
  southern       SO    Suan Chin           5.1  .95  4  15
  southeast      SE    Patricia Hemenway   4.0  .7   4  17
  eastern        EA    TB Savage           4.4  .84  5  20
  northeast      NE    AM Main Jr.         5.1  .94  3  13
  north          NO    Margot Weber        4.5  .89  5  9
  central        CT    Ann Stephens        5.7  .94  5  13
```

说明

y 命令把第 1~3 行中所有的小写字母转换成大写。正则表达式元字符对 y 命令不起作用。与替换分隔符一样，斜杠可以被替换成其他的字符。

5.9.13 退出：q 命令

q 命令表示退出命令。该命令将导致 sed 程序退出，且不再进行其他的处理。

```
% cat datafile
northwest      NW    Charles Main        3.0  .98  3  34
western        WE    Sharon Gray         5.3  .97  5  23
southwest      SW    Lewis Dalsass       2.7  .8   2  18
southern       SO    Suan Chin           5.1  .95  4  15
southeast      SE    Patricia Hemenway   4.0  .7   4  17
eastern        EA    TB Savage           4.4  .84  5  20
northeast      NE    AM Main Jr.         5.1  .94  3  13
north          NO    Margot Weber        4.5  .89  5  9
central        CT    Ann Stephens        5.7  .94  5  13
```

范例 5-31

```
% sed '5q' datafile
northwest         NW        Charles Main          3.0   .98   3   34
western           WE        Sharon Gray           5.3   .97   5   23
southwest         SW        Lewis Dalsass         2.7   .8    2   18
southern          SO        Suan Chin             5.1   .95   4   15
southeast         SE        Patricia Hemenway     4.0   .7    4   17
```

说明

打印完第 5 行之后，q 命令让 sed 程序退出。

范例 5-32

```
% sed '/Lewis/{ s/Lewis/Joseph/;q; }' datafile
northwest         NW        Charles Main          3.0   .98   3   34
western           WE        Sharon Gray           5.3   .97   5   23
southwest         SW        Joseph Dalsass        2.7   .8    2   18
```

说明

在某行匹配到模式 Lewis 时，s 表示先用 Joseph 替换 Lewis，然后 q 命令让 sed 程序退出。

5.9.14 暂存和取用：h 命令和 g 命令

h 命令表示暂存，g 命令表示取用。当从文件中读取一行时，sed 将该行放置到模式缓冲区中，并在该行上执行命令。有一个称为暂存缓冲区的附加缓冲区(两个缓存区都能够存储 8,192 个字节)。h 命令将一行从模式缓冲区发送到暂存缓冲区中，随后通过 g 或者 G 命令获取。

% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 5-33

```
% sed -e '/northeast/h'  -e  '$G'  datafile
    northwest         NW        Charles Main          3.0   .98   3   34
    western           WE        Sharon Gray           5.3   .97   5   23
    southwest         SW        Lewis Dalsass         2.7   .8    2   18
```




```
    southern          SO      Suan Chin           5.1   .95   4   15
    southeast         SE      Patricia Hemenway   4.0   .7    4   17
    eastern           EA      TB Savage           4.4   .84   5   20
--> northeast         NE      AM Main Jr.         5.1   .94   3   13
    north             NO      Margot Weber        4.5   .89   5   9
    central           CT      Ann Stephens        5.7   .94   5   13
--> northeast         NE      AM Main Jr.         5.1   .94   3   13
```

说明

sed 处理文件时，会把文件的每一行都保存在一个临时缓冲区中，该缓冲区被称为模式空间。sed 处理完每一行后，都会将其打印在屏幕上，除非该行被删除或取消打印。之后，模式空间被清空，下一输入行被保存进来等待处理。本例中，包含模式 northeast 的行被找到之后，就被保存在模式空间里，h 命令把它复制并保存到另一个特殊的缓冲区，这个特殊的缓冲区称为暂存缓冲区(holding buffer)。在第二条 sed 指令中，sed 读到最后一行($)时，G 命令指示它从暂存缓冲区中读一行，将其放回模式空间缓冲区，追加在模式空间内当前行(本例中是最后一行)的后面。简而言之：所有包含模式 northeast 的行都被复制并追加到文件尾部。

范例 5-34

```
% sed -e '/WE/{h; d; }' -e '/CT/{G; }' datafile
    northwest         NW      Charles Main        3.0   .98   3   34
    southwest         SW      Lewis Dalsass       2.7   .8    2   18
    southern          SO      Suan Chin           5.1   .95   4   15
    southeast         SE      Patricia Hemenway   4.0   .7    4   17
    eastern           EA      TB Savage           4.4   .84   5   20
    northeast         NE      AM Main Jr.         5.1   .94   3   13
    north             NO      Margot Weber        4.5   .89   5   9
    central           CT      Ann Stephens        5.7   .94   5   13
--> western           WE      Sharon Gray         5.3   .97   5   23
```

说明

如果找到模式 WE，h 命令将从模式空间复制行到暂存缓冲区(稍后会用 G 或 g 命令重新提取该行)。本例中，包含模式 WE 的行被找到后先存在模式空间里，接着 h 命令会复制它并保存在暂存缓冲区中。之后，d 命令将该行从模式空间删除，即不再显示它。第二条命令将在行中查找 CT，找到模式 CT 后，g 命令取出之前保存在暂存缓冲区中的副本，并追加在模式空间内当前行的后面。简而言之：所有包含模式 WE 的行都被移出原位置，移到包含模式 CT 的行后面，可参见下一小节“暂存和互换：h 和 x 命令”中的内容。

范例 5-35

```
% sed -e '/northeast/h'  -e  '$g'  datafile
    northwest         NW      Charles Main        3.0   .98   3   34
    western           WE      Sharon Gray         5.3   .97   5   23
    southwest         SW      Lewis Dalsass       2.7   .8    2   18
    southern          SO      Suan Chin           5.1   .95   4   15
    southeast         SE      Patricia Hemenway   4.0   .7    4   17
    eastern           EA      TB Savage           4.4   .84   5   20
```

```
──► northeast      NE     AM Main Jr.       5.1   .94   3   13
    north          NO     Margot Weber      4.5   .89   5   9
──► northeast      NE     AM Main Jr.       5.1   .94   3   13
```

说明

sed 处理文件时，会把文件的每一行都保存在一个称为模式空间的临时缓冲区中。sed 处理完每一行后，都会将其打印在屏幕上，除非该行被删除或取消打印。之后，模式空间被清空，下一输入行被保存进来等待处理。本例中，包含模式 northeast 的行被找到之后，就被保存在模式空间里，h 命令把它复制并保存到另一个称为暂存缓冲区的特殊缓冲区中。在第二条 sed 指令中，sed 读到最后一行($)时，g 命令指示它从暂存缓冲区中读入一行，并将其放回模式空间缓冲区，替换掉模式空间内当前行(本例中是最后一行)。简而言之：包含模式 northeast 的行被复制并被用来替换文件的最后一行。

% cat datafile

northwest	NW	Charles Main	3.0	.98	3	34
western	WE	Sharon Gray	5.3	.97	5	23
southwest	SW	Lewis Dalsass	2.7	.8	2	18
southern	SO	Suan Chin	5.1	.95	4	15
southeast	SE	Patricia Hemenway	4.0	.7	4	17
eastern	EA	TB Savage	4.4	.84	5	20
northeast	NE	AM Main Jr.	5.1	.94	3	13
north	NO	Margot Weber	4.5	.89	5	9
central	CT	Ann Stephens	5.7	.94	5	13

范例 5-36

```
% sed -e '/WE/{h; d; }' -e '/CT/{g; }' datafile
northwest        NW     Charles Main       3.0   .98   3   34
southwest        SW     Lewis Dalsass      2.7   .8    2   18
southern         SO     Suan Chin          5.1   .95   4   15
southeast        SE     Patricia Hemenway  4.0   .7    4   17
eastern          EA     TB Savage          4.4   .84   5   20
northeast        NE     AM Main Jr.        5.1   .94   3   13
north            NO     Margot Weber       4.5   .89   5   9
western          WE     Sharon Gray        5.3   .97   5   23
```

说明

如果在找到了模式 WE，h 命令就将该行从模式缓冲区复制到一个暂存缓冲区。d 命令将模式空间中的行删除。若匹配了模式 CT，则 g 命令将获取暂存缓冲区中的备份并覆盖模式空间中的当前行。简而言之：所有包含 WE 模式的行将移到包含 CT 的行上，并进行覆盖(参见图 5-1)。



/WE/{h; d;}

western	WE	Sharon Gray	5.3	.97	5	23

h：若找到 WE，则模式空间(上)中的内容将被复制到暂存缓冲区(下)里并由 d 命令清空

western	WE	Sharon Gray	5.3	.97	5	23

/CT/{g; }

central	CT	Ann Stephens	5.7	.94	5	13

g：若找到 CT，则模式空间(上)中的内容将被暂存缓冲区(下)中的覆盖并显示下面的内容

western	WE	Sharon Gray	5.3	.97	5	23

图 5-1　模式空间和暂存缓冲区。参见范例 5-36

5.9.15　暂存和互换：h 命令和 x 命令

x 命令表示互换。sed 使用该命令互换暂存缓冲区中的模式与模式缓冲区中的当前行。

范例 5-37

```
% sed -e '/Patricia/h'  -e '/Margot/x' datafile
northwest       NW      Charles Main        3.0   .98   3   34
western         WE      Sharon Gray         5.3   .97   5   23
southwest       SW      Lewis Dalsass       2.7   .8    2   18
southern        SO      Suan Chin           5.1   .95   4   15
southeast       SE      Patricia Hemenway   4.0   .7    4   17
eastern         EA      TB Savage           4.4   .84   5   20
northeast       NE      AM Main Jr.         5.1   .94   3   13
southeast       SE      Patricia Hemenway   4.0   .7    4   17
central         CT      Ann Stephens        5.7   .94   5   13
```

说明

x 命令将暂存缓冲区的内容和模式空间内的当前行互换(交换)。sed 找到包含模式 Patricia 的行后，会将其保存在暂存缓冲区中。然后，当它找到包含模式 Margot 的行时，就将模式空间中的内容与暂存缓冲区中的互换。简而言之：包含 Margot 的行将被包含 Patricia 的行替换。

5.10　sed 脚本编程

sed 脚本就是写在文件中的一列 sed 命令。在命令行启动 sed 命令时，如果想让 sed 知道这些命令来自文件，就要用-f 选项带上 sed 脚本的文件名。sed 对脚本中命令行书写方式的要求很特别。它要求命令的末尾不能有任何多余的空格或文本。如果命令不能独占一行，就必须以分号结尾。执行脚本时，sed 先将输入文件中第一行复制到模式缓冲区，然后对其执行脚本中的所有命令。每一行处理完毕后，sed 再复制文件中下一行到模式缓冲区，对其执行脚本中的所有命令。如果脚本中有语法错误，sed 的运行就会错乱。

在命令行使用 sed 时常常会涉及到和 shell 的交互，而使用 sed 脚本则完全不必为此操心。不再用引号来确保 sed 命令不被 shell 解释，而且只用一个反斜杠就起到续行的作用。事实上，在 sed 脚本中根本就不能使用引号，除非它们是查找串的一部分。

下面是范例使用的 datafile 文件。

```
% cat datafile
northwest    NW    Charles Main         3.0    .98    3    34
western      WE    Sharon Gray          5.3    .97    5    23
southwest    SW    Lewis Dalsass        2.7    .8     2    18
southern     SO    Suan Chin            5.1    .95    4    15
southeast    SE    Patricia Hemenway    4.0    .7     4    17
eastern      EA    TB Savage            4.4    .84    5    20
northeast    NE    AM Main Jr.          5.1    .94    3    13
north        NO    Margot Weber         4.5    .89    5    9
central      CT    Ann Stephens         5.7    .94    5    13
```

5.10.1 sed 脚本范例

范例 5-38

```
% cat sedding1              # Look at the contents of the sed script
1  # My first sed script by Jack Sprat
2  /Lewis/a\
3     Lewis is the TOP Salesperson for April!!\
      Lewis is moving to the southern district next month.\
4     CONGRATULATIONS!
5  /Margot/c\
      *******************\
      MARGOT HAS RETIRED\
      ********************
6  1i\
   EMPLOYEE DATABASE\
   ---------------------
7  $d

% sed -f sedding1 datafile   # Execute the sed script commands; input file
is datafile
EMPLOYEE DATABASE
- - - - - - - - - - - - - - - - - - - - - -
northwest         NW      Charles Main         3.0   .98    3    34
western           WE      Sharon Gray          5.3   .97    5    23
southwest         SW      Lewis Dalsass        2.7   .8     2    18
Lewis is the TOP Salesperson for April!!
Lewis is moving to the southern district next month.
CONGRATULATIONS!
southern          SO      Suan Chin            5.1   .95    4    15
southeast         SE      Patricia Hemenway    4.0   .7     4    17
eastern           EA      TB Savage            4.4   .84    5    20
```

```
northeast            NE        AM Main Jr.            5.1  .94  3   13

********************
MARGOT HAS RETIRED
*****************
```

说明

1. 这一行是注释。注释必须独占一行并以#开头。

2. 如果某一行包含模式 Lewis，就将下面这三行加到它后面。

3. 所追加的内容中，除最后一行外，每一行都以反斜杠结尾。反斜杠后必须紧跟换行符；如果换行符后有多余的文字，哪怕只是空格，sed 都会报错。

4. 所追加的最后一行内容不需要以反斜杠结尾，于是 sed 就知道这是要追加的最后一行，下一行便属于另一条命令了。

5. 所有包含模式 Margot 的行都会被下面这三行文字替换(c 命令)。

6. 下面这两行被插到第一行的前面(i 命令)。

7. 最后一行($)被删除。

范例 5-39

```
% cat sedding2              # Look at the contents of the sed script
# This script demonstrates the use of curly braces to nest addresses
# and commands. Comments are preceded by a pound sign (#) and must
# be on a line by themselves. Commands are terminated with a newline
# or semicolon.  If there is any text after a command, even one
# space, you receive an error message:
#     sed: Extra text at end of command:

1  /western/, /southeast/{
       /^ *$/d
       /Suan/{ h; d; }
   }
2  /Ann/g
3  s/TB \(Savage\)/Thomas \1/
4 % sed -f sedding2 datafile
northwest            NW     Charles Main        3.0   .98    3    34
western              WE     Sharon Gray         5.3   .97    5    23
southwest            SW     Lewis Dalsass       2.7   .8     2    18
southeast            SE     Patricia Hemenway   4.0   .7     4    17
eastern              EA     Thomas Savage       4.4   .84    5    20
northeast            NE     AM Main Jr.         5.1   .94    3    13
north                NO     Margot Weber        4.5   .89    5    9
southern             SO     Suan Chin           5.1   .95    4    15
```

说明

1. 对从 western 到 southcast 之间的各行执行以下操作：删除空行，将匹配 Suan 的行先从模式缓冲区复制到暂存缓冲区，然后从模式缓冲区中删除。

2. 当匹配模式 Ann 时，g 命令将暂存缓冲区内的行复制到模式缓冲区里，从而覆盖模

式缓冲区中当前的内容。

3. 各行中出现的模式 TB Savage 都被替换为 Thomas 和被标记的模式 Savage。查找串中，Savage 被括在转义的圆括号中，这样被括起来的字符串就成为标签，可以被引用。Savage 是这个示例的第一个标签，因此可以用\1 来引用它。

5.10.2 回顾

表 5-4 列出了一些 sed 命令和它们的功能。

表 5-4 sed 命令回顾

命 令	功 能
sed –n '/sentimental/p' filex	把文件 filex 中所有包含 sentimental 的行打印在屏幕上 filex 的内容不会被改变 如果没有-n 选项，所有包含 sentimental 的行都会被打印两次
sed '1,3d' filex > newfilex	删除文件 filex 的前 3 行，将修改结果保存到 newfilex 文件中
sed '/[Dd]aniel/d' filex	删除包含 Daniel 或 daniel 的行
sed –n '15,20p' filex	只打印第 15~20 行
sed '1,10s/Montana/MT/g' filex	将第 1~10 行的所有 Montana 全局替换为 MT
sed '/March/\!d' filex (csh)	删除所有不含 March 的行。(只有在 csh 中才要用反斜杠来转义历史字符)
sed '/March/!d' filex (sh)	
sed '/report/s/5/8' filex	把所有包含 report 的行里出现的第一个 5 改为 8
sed 's/….//' filex	删除每行的前 4 个字符
sed 's/…$//' filex	删除每行的后 3 个字符
sed '/east/,/west/s/North/South/' filex	把从 east 到 west 这个范围内所有行中出现的 North 替换为 South
sed –n '/Time off/w timefile' filex	将所有包含 Time off 的行写入到 timefile 文件中
sed 's/\([Oo]ccur\)ence/\1rence/' file	将所有 Occurence 替换成 Occurrence。occurrence 替换成 Occurence
sed –n 'l' filex	打印所有以\nn 显示非打印字符(nn 是字符的十进制值)，制表键(tab)显示为>的行

习题 2 sed 的使用练习

(参考从本书合作站点下载的文件中名为 datebook 的文件)

Steve Blenheim:238-923-7366:95 Latham Lane, Easton, PA 83755:11/12/56:20300
Betty Boop:245-836-8357:635 Cutesy Lane, Hollywood, CA 91464:6/23/23:14500
Igor Chevsky:385-375-8395:3567 Populus Place, Caldwell, NJ 23875:6/18/68:23400
Norma Corder:397-857-2735:74 Pine Street, Dearborn, MI 23874:3/28/45:245700
Jennifer Cowan:548-834-2348:583 Laurel Ave., Kingsville, TX 83745:10/1/35:58900
Jon DeLoach:408-253-3122:123 Park St., San Jose, CA 04086:7/25/53:85100
Karen Evich:284-758-2857:23 Edgecliff Place, Lincoln, NB 92743:7/25/53:85100



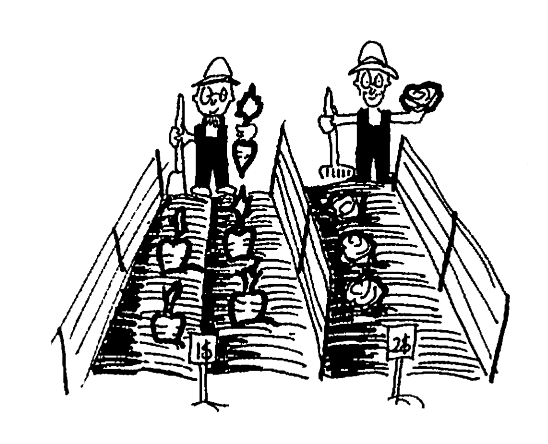
Karen Evich:284-758-2867:23 Edgecliff Place, Lincoln, NB 92743:11/3/35:58200
Karen Evich:284-758-2867:23 Edgecliff Place, Lincoln, NB 92743:11/3/35:58200
Fred Fardbarkle:674-843-1385:20 Parak Lane, Duluth, MN 23850:4/12/23:780900
Fred Fardbarkle:674-843-1385:20 Parak Lane, Duluth, MN 23850:4/12/23:780900
Lori Gortz:327-832-5728:3465 Mirlo Street, Peabody, MA 34756:10/2/65:35200
Paco Gutierrez:835-365-1284:454 Easy Street, Decatur, IL 75732:2/28/53:123500
Ephram Hardy:293-259-5395:235 CarltonLane, Joliet, IL 73858:8/12/20:56700
James Ikeda:834-938-8376:23445 Aster Ave., Allentown, NJ 83745:12/1/38:45000
Barbara Kertz:385-573-8326:832 Ponce Drive, Gary, IN 83756:12/1/46:268500
Lesley Kirstin:408-456-1234:4 Harvard Square, Boston, MA 02133:4/22/62:52600
William Kopf:846-836-2837:6937 Ware Road, Milton, PA 93756:9/21/46:43500
Sir Lancelot:837-835-8257:474 Camelot Boulevard, Bath, WY 28356:5/13/69:24500
Jesse Neal:408-233-8971:45 Rose Terrace, San Francisco, CA 92303:2/3/36:25000
Zippy Pinhead:834-823-8319:2356 Bizarro Ave., Farmount, IL 84357:1/1/67:89500
Arthur Putie:923-835-8745:23 Wimp Lane, Kensington, DL 38758:8/31/69:126000
Popeye Sailor:156-454-3322:945 Bluto Street, Anywhere, USA 29358:3/19/35:22350
Jose Santiago:385-898-8357:38 Fife Way, Abilene, TX 39673:1/5/58:95600
Tommy Savage:408-724-0140:1222 Oxbow Court, Sunnyvale, CA 94087:5/19/66:34200
Yukio Takeshida:387-827-1095:13 Uno Lane, Ashville, NC 23556:7/1/29:57000
Vinh Tranh:438-910-7449:8235 Maple Street, Wilmington, VM 29085:9/23/63:68900

1. 把 Jon 的名字改为 Jonathan。
2. 删除头 3 行。
3. 打印第 5~10 行。
4. 删除含有 Lane 的所有行。
5. 打印所有生日在十一月或十二月的行。
6. 在以 Ered 开头的各行末尾加上 3 颗星。
7. 将所有包含 Jose 的行都替换为 JOSE HAS RETIRED。
8. 把 Popeye 的生日改为 11/14/46，假定您不知道 Popeye 的生日，设法用正则式查找出来。
9. 删除所有空行。
10. 写一个能完成下列任务的 sed 脚本。
    a) 在第一行上方插入标题 PERSONNEL FILE。
    b) 删除以 500 结尾的工资项。
    c) 把名和姓的位置颠倒后，打印文件内容。
    d) 在文件末尾加上 THE END。

chapter



awk 实用程序



6.1 什么是 awk、nawk、gawk

awk 是一种用于处理数据和生成报告的 UNIX 编程语言。nawk 是 awk 的新版本，gwak 是基于 Linux 的 GNU 版本。

处理的数据可以来自标准输入、一个或多个文件，也可以来自某个进程的输出。awk 可以在命令行进行一些简单的操作，也可以编写成程序来处理较大的应用。因为 awk 可以处理数据，所以它是执行 shell 脚本和管理小型数据库不可或缺的工具。

awk 以逐行方式扫描文件(或输入)，从第一行到最后一行，以查找匹配某个特定模式的文本行，并对这些文本行执行(括在花括号中的)指定动作。如果只给出模式而未指定动作，则所有匹配该模式的行都显示在屏幕上；如果只指定动作而未定义模式，会对所有输入行执行指定动作。

6.1.1 awk 简介

awk 是 3 个姓氏的首字母，代表创建该语言的 3 位作者：Alfred Aho、Brian Kernighan 和 Peter Weinberger。当然也可以叫它 wak 或 kaw，但常用的是 awk。

6.1.2 awk 版本

awk 的版本有很多，包括：旧版 awk、新版 awk、GNU awk (gawk)、POSIX awk 等。awk 最初编写于 1977 年，该版本在 1985 年得以改进以支持更大的程序，还增加了可用户自定义函数、动态正则表达式、同时处理多个输入文件等功能。在大多数系统上，如果使用旧版本，应输入命令 awk，新版本的命令是 nawk，GNU 版的命令则是 gawk[①]。

① 在 SCO UNIX 上，新版本的命令是 awk。而 Linux 系统上，GNU 版本的却是命令 awk。本文主要针对新 awk，即 nawk，进行介绍。GNU 实现的 gawk 与 nawk 完全向上兼容。



6.2 awk 的格式

awk 程序由 awk 命令、括在引号(或写在文件)中的程序指令以及输入文件的文件名几个部分组成。如果没有指定输入文件，输入则来自标准输入(stdin)，即键盘。

awk 指令由模式、操作、或模式与操作的组合组成。模式是由某种类型的表达式组成的语句。如果某个表达式中没有出现关键字 if，但实际计算时却暗含 if 这个词，那么，这个表达式就是模式。操作由括在大括号内的一条或多条语句组成，语句之间用分号或换行符隔开。模式则不能被括在大括号中，模式由括在两个正斜杠之间的正则表达式、一个或多个 awk 操作符组成的表达式组成。

awk 命令可以在命令行输入，也可以写在 awk 脚本文件里。要处理的文本行则来自文件、管道或标准输入。

6.2.1 从文件输入

下面这些例子中的百分号(%)是 shell 的命令提示符。请注意：下例中都采用了 nawk 命令，如果所在系统为 HP-UX 系统或者是 Linux 系统则应该采用相应的 awk 或 gawk 命令。

格式

```
% nawk 'pattern' filename
% nawk '{action}' filename
% nawk 'pattern {action}' filename
```

下面演示一个名为 employees 的文件。

范例 6-1

```
% cat employees
Tom Jones          4424    5/12/66          543354
Mary Adams         5346    11/4/63          28765
Sally Chang        1654    7/22/54          650000
Billy Black        1683    9/23/44          336500

% nawk '/Mary/'  employees
Mary Adams         5346    11/4/63          28765
```

说明

nawk 打印出所有包含模式 Mary 的行。

范例 6-2

```
% cat employees
Tom Jones          4424    5/12/66          543354
Mary Adams         5346    11/4/63          28765
Sally Chang        1654    7/22/54          650000
Billy Black        1683    9/23/44          336500

% nawk '{print $1}' employees
Tom
```

```
Mary
Sally
Billy
```

说明

nawk 打印出文件 employees 的第一个字段，字段从行的左端开始，以空白符分隔。

范例 6-3

```
% cat employees
Tom Jones          4424    5/12/66          543354
Mary Adams         5346    11/4/63          28765
Sally Chang        1654    7/22/54          650000
Billy Black        1683    9/23/44          336500

% nawk '/Sally/{print $1, $2}' employees
Sally Chang
```

说明

当文件 employees 中的某一行含有模式 Sally 时，nawk 打印该行的头两个字段。记住，字段的分隔符是空白符。

6.2.2 从命令输入

可以将一条或多条 UNIX 或者 Linux 命令的输出通过管道发给 awk 处理。shell 程序通常使用 awk 来处理命令。

格式

```
% command | nawk 'pattern'
% command | nawk '{action}'
% command | nawk 'pattern {action}'
```

范例 6-4

```
1   % df | nawk '$4 > 75000'
    /oracle   (/dev/dsk/c0t0d057 ):390780 blocks     105756 files
    /opt     (/dev/dsk/c0t0d058   ):1943994 blocks     49187 files

2   % rusers  | nawk '/root$/{print  $1}'
    owl
    crow
    bluebird
```

说明

1. df 命令报告文件系统的磁盘空间剩余情况。df 命令的输出通过管道发给 nawk(新 awk)。如果其中某行的第 4 个字段大于 75000(块)，则该行被打印。

2. rusers 命令打印出有哪些用户登录在局域网的其他机器上。rusers 的输出通过管道发送给 awk，作为它的输入。如果输出中有某行以正则式 root 结尾($)，就打印该行的第一个字段，即 root 登录的所有机器名。



6.3 awk 工作原理

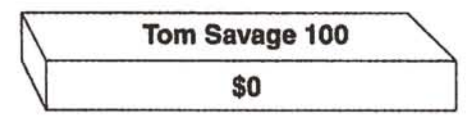
在详细阐述 awk 之前，我们首先了解一下 awk 是如何完成自己的任务的，我们将以一个名为 names(仅有三行记录)的文件为例进行说明。

```
Tom Savage 100
Molly Lee 200
John Doe 300
```

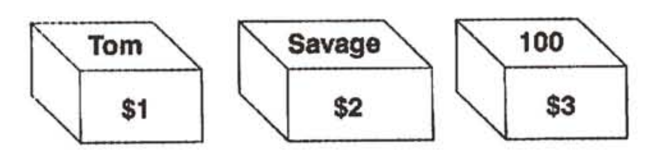
使用下面的 awk 命令：

```
% nawk '{print $1, $3}' names
```

(1) awk 使用一行作为输入(通过文件或者管道)，并将这一行赋给内部变量$0，默认时每一行也可以称为一个记录，以换行符结束。



(2) 然后，行被空格分解成字段(单词)，每一个字段存储在已编号的变量中，从$1 开始，可以多达 100 个字段。



(3) awk 如何知道空格是用来分隔字段的呢？因为有另一个内部变量 FS 用来确定字段的分隔符。初始时，FS 被赋为空格——包含制表符和空格符。如果需要使用其他的字符分隔字段，如冒号或破折号，则需要将 FS 变量的值设为新的字段分隔符(参见 6.6.3 节，“字段分隔符”)。

(4) awk 打印字段时，将以下面方式使用 print 函数。

```
{print $1,$3}
```

输出显示了每个字段使用空格进行分隔，如下所示。

```
Tom 100
Molly 200
John 300
```

awk 在 Tom 和 100 之间加入了空格，因为在$1 和$3 之间存在一个逗号。逗号比较特殊，它映射为另一个内部变量，称为输出字段分隔符(output field separator，OFS)，OFS 默认为空格。逗号被 OFS 变量中存储的字符替换。

(5) awk 输出之后，将从文件中获取另一行，并将其存储到$0 中，覆盖原来的内容，然后将新的字符串分隔成字段并进行处理。这个过程将持续到整个文件的所有行都处理完毕。

6.4 格式化输出

6.4.1 print 函数

awk 命令的操作部分被括在大括号内。如果未指定操作，则匹配到模式时，awk 会采取默认操作，即在屏幕上打印包含模式的行。print 函数用于打印不需要特别编排格式的简单输出。更为复杂的格式编排则要使用 printf 和 sprintf 函数。如果熟悉 C 语言，那么一定懂得如何使用 printf 和 sprintf。

也可以用{print}形式在 awk 命令的动作部分显式地调用 print 函数。print 函数的参数可以是变量、数值或字符串常量。字符串必须用双引号括起来。参数之间用逗号分隔，如果没有逗号，所有的参数就会被串在一起。逗号等价于 OFS 中的值，默认情况下是空格。

print 函数的输出可以被重定向，也可以通过管道传给另一个程序。其他程序的输出也可以通过管道交给 awk 打印(参见第 1.6 节中的“重定向”和“管道”)。

范例 6-5

```
% date
Wed Jul 28 22:23:16 PDT 2004

% date | nawk '{ print "Month: " $2 "\nYear: " , $6 }'
Month: Jul
Year: 2004
```

说明

UNIX 中，date 命令的输出经管道发送给 nawk。打印显示为字符串 Month:，后面跟 date 输出结果中的第 2 个字段，然后是另一个字符串，该串中包含换行符\n 和 Year:，最后是 date 输出结果中的第 6 个字段($6)。

转义序列 转义序列用一个反斜杠后跟一个字母或数字来表示。它们可以用在字符串中，代表制表符、换行符、换页符等(参见表 6-1)。

表 6-1 print 函数使用的转义序列

转义序列	含义
\b	退格
\f	换页
\n	换行
\r	回车
\t	制表符
\047	八进制值 47，即单引号
\c	C 代表任一其他字符，例如“\”



范例 6-6

```
Tom Jones       4424    5/12/66    543354
Mary Adams      5346    11/4/63    28765
Sally Chang     1654    7/22/54    650000
Billy Black     1683    9/23/44    336500

% nawk '/Sally/{print "\t\tHave a nice day, " $1, $2 "\!"}' employees
        Have a nice day, Sally Chang!
```

说明

如果某一行包含模式 Sally，print 函数就会打印出两个制表符，字符串“Have a nice day”，该行的第一个字段($1，本例中是 Sally)和第二个字段($2，本例中是 Chang)，后面跟一个带感叹号的字符串。

6.4.2 OFMT 变量

打印数字时，可能需要控制数字的格式。这可以通过 printf 函数来实现，但是，通过设置一个特殊的 awk 变量 OFMT，使用 print 函数时也可以控制数字的打印格式。OFMT 的默认值是“%.6gd”，表示只打印小数部分的前 6 位(下一节将会介绍如何修改 OFMT 的值)。

范例 6-7

```
% nawk 'BEGIN{OFMT="%.2f"; print 1.2456789, 12E-2}'
1.25  0.12
```

说明

如果设置了变量 OFMT，在打印浮点数时，就只打印小数部分的前两位。百分号(%)表示接下来要定义格式。

6.4.3 printf 函数

打印输出时，可能需要指定字段间的空格数，从而把列排整齐。在 print 函数中使用制表符并不能保证得到想要的输出，因此，可以用 printf 函数来格式化特别的输出。

printf 函数返回一个带格式的字符串给标准输出，如同 C 语言中的 printf 语句一样。printf 语句包括一个加引号的控制串，控制串中可能嵌有若干格式说明和修饰符。控制串后面跟一个逗号，之后是一列由逗号分隔的表达式。printf 函数根据控制串中的说明编排这些表达式的格式。与 print 函数不同的是，printf 不会在行尾自动换行。因此，如果要换行，就必须在控制串中提供转义字符\n。

每一个百分号和格式说明都必须有一个对应的变量。要打印百分号就必须在控制串中给出两个百分号。请参考表 6-2 列出的 printf 转义字符和表 6-3 中的 printf 修饰符。格式说明由百分号引出，表 6-4 列出了 printf 所用的格式说明符。

表 6-2 printf 使用的转义字符

转义字符	定义
c	字符
s	字符串
d	十进制整数
ld	十进制长整数
u	十进制无符号整数
lu	十进制无符号长整数
x	十六进制整数
lx	十六进制长整数
o	八进制整数
lo	八进制长整数
e	用科学记数法(e 记数法)表示的浮点数
f	浮点数
g	选用 e 或 f 中较短的一种形式

表 6-3 printf 的修饰符

字符	定义
-	左对齐修饰符
#	显示 8 进制整数时在前面加个 0 显示 16 进制整数时在前面加 0x
+	显示使用 d、e、f 和 g 转换的整数时，加上正负号+或-
0	用 0 而不是空白符来填充所显示的值

表 6-4 printf 的格式说明符

格式说明符	功能
假定 x = 'A'、y = 15、z = 2.3 且$1 = Bob Smith。	
%c	打印单个 ASCII 字符 printf("The character is %c\n", x) 输出：The character is A
%d	打印一个十进制数 printf("The boy is %d years old\n", y) 输出：The boy is 15 years old
%e	打印数字的 e 记数法形式 printf("z is %e\n", z)打印：z is 2.3e+01
%f	打印一个浮点数 printf("z is %f\n", 2.3 * 2) 输出：z is 4.600000



(续表)

格式说明符	功　能
%o	打印数字的八进制值 printf("y is %o\n", y) 输出：z is 17
%s	打印一个字符串 printf("The name of the culprit is %s\n", $1) 输出：The name of the culprit is Bob Smith
%x	打印数字的十六进制值 printf("y is %x\n", y) 输出：x is f

打印变量时，输出所在的位置称为“域”(field)，域的宽度(width)是指该域中所包含的字符个数。

下面这些例子中，printf 控制串里的管道符(竖杠)是文本的一部分，用于指示格式的起始与结束。

范例 6-8

```
1  % echo "UNIX" | nawk ' {printf "|%-15s|\n", $1}'
   (Output)
   |UNIX           |

2  % echo "UNIX" | nawk '{ printf "|%15s|\n", $1}'
   (Output)
   |           UNIX |
```

说明

1. 对于 echo 命令的输出，UNIX 是经管道发给 nawk。printf 函数包含一个控制串。百分号让 printf 做好准备，它要打印一个占 15 个格、向左对齐的字符串，这个字符串夹在两个竖杠之间，并且以换行符结尾。百分号后的短划线表示左对齐。控制串后面跟了一个逗号和$1。printf 将根据控制串中的格式说明来格式化字符串 UNIX。

2. 字符串 UNIX 被打印成一个占 15 格、向右对齐的字符串，夹在两个竖杠之间，以换行符结尾。

范例 6-9

```
% cat employees
Tom Jones       4424    5/12/66    543354
Mary Adams      5346    11/4/63    28765
Sally Chang     1654    7/22/54    650000
Billy Black     1683    9/23/44    336500

% nawk '{printf "The name is: %-15s ID is %8d\n", $1, $3}' employees
The name is Tom                ID is 4424
```

```
The name is Mary          ID is 5346
The name is Sally         ID is 1654
The name is Billy         ID is 1683
```

说明

要打印的字符串放置在两个双引号之间。第一个格式说明符是%-15s，它对应的参数是$1，紧挨着控制串的右半边引号后面的那个逗号。百分号引出格式说明：短划线表示左对齐，15s 表示占 15 格的字符串。这条命令用来打印一个左对齐、占 15 格的字符串，后面跟着字符串的 ID 和一个整数。

格式%8d 表示在字符串的这个位置打印$2 的十进制(整数)值。这个整数占 8 格，向右对齐。您也可以选择将加引号的字符串和表达式放在圆括号里。

6.5 文件中的 awk 命令

如果 awk 命令被写在文件里，就要用-f 选项指定 awk 的文件名，后面再加上所要处理的输入文件的文件名。awk 从缓冲区读入一条记录，接着测试 awk 文件中的每一条命令，然后对读入的记录执行命令。处理完第一条记录后，awk 将其丢弃，接着将下一条记录读入缓冲区，依次处理所有记录。如果没有模式限制，默认的操作就是打印全部记录。而模式如果没有相应的操作，则默认行为是打印匹配它的记录。

范例 6-10

(数据库)
```
  $1       $2       $3     $4        $5
  Tom      Jones    4424   5/12/66   543354
  Mary     Adams    5346   11/4/63   28765
  Sally    Chang    1654   7/22/54   650000
  Billy    Black    1683   9/23/44   336500

% cat awkfile
1  /^Mary/{print "Hello Mary!"}
2  {print $1, $2, $3}

% nawk -f awkfile employees
  Tom Jones 4424
  Hello Mary!
  Mary Adams 5346
  Sally Chang 1654
  Billy Black 1683
```

说明

1. 如果记录以正则表达式 Mary 开头，则打印字符串“Hello Mary!”。操作取决于它前面的模式是否匹配。且打印的字段之间以空白符分隔。



2. 打印每条记录的第 1、第 2、第 3 字段。awk 对每行都执行该操作，因为没有限制该操作的模式。

6.6 记录与字段

6.6.1 记录

在 awk 看来输入数据具有格式和结构，而不是无休止的字符串。默认情况下，每一行称为一条记录，以换行符结束。

记录分隔符 默认情况下，输入和输出记录的分隔符(行分隔符)都是回车符(换行符)，分别保存在 awk 的内置变量 ORS 和 RS 中。ORS 和 RS 的值可以修改，但是只能以特定方式进行修改。

变量$0 awk 用$0 指代整条记录(当$0 因替换或赋值而改变时，NF 的值，即字段的数目值也可能改变)。换行符的值保存在 awk 的内置变量 RS 中，其默认值为回车。

范例 6-11

```
% cat employees
Tom Jones       4424     5/12/66     543354
Mary Adams      5346     11/4/63     28765
Sally Chang     1654     7/22/54     650000
Billy Black     1683     9/23/44     336500

% nawk '{print $0}' employees
Tom Jones       4424     5/12/66     543354
Mary Adams      5346     11/4/63     28765
Sally Chang     1654     7/22/54     650000
Billy Black     1683     9/23/44     336500
```

说明

nawk 的变量$0 保存当前记录的内容。它被打印在屏幕上。默认情况下，下面这条命令也可以打印 nawk 记录：

```
% nawk '{printf}' employees
```

变量 NR 每条记录的记录号都保存在 awk 的内置变量 NR 中。每处理完一条记录，NR 的值都会加 1。

范例 6-12

```
% cat employees
Tom Jones       4424     5/12/66     543354
Mary Adams      5346     11/4/63     28765
Sally Chang     1654     7/22/54     650000
Billy Black     1683     9/23/44     336500

% nawk '{print NR, $0}'  employees
1 Tom Jones       4424     5/12/66     543354
2 Mary Adams      5346     11/4/63     28765
```

```
3 Sally Chang     1654     7/22/54     650000
4 Billy Black     1683     9/23/44     336500
```

说明

照文件原样打印出每一条记录，并且在行首加上了它的记录号(NR)。

6.6.2 字段

每条记录都是由称为字段(field)的词组成，默认情况下，字段间用空白符(即空格或制表符)分隔。每个这样的词称为一个字段，awk 在内置变量 NF 中保存记录的字段数。NF 的值因行而异，其上限与具体版本的实现相关，通常是每行最多 100 个字段。可以创建新的字段。下面这个例子中有 4 条记录(行)和 5 个字段(列)。每条记录都是从第一个字段(用 $1 表示)开始，然后是第二个字段($2)，以此类推。

范例 6-13

```
(字段由美元符号($)和字段编号做了标示)
(数据库)
$1       $2        $3      $4         $5
Tom      Jones     4424    5/12/66    543354
Mary     Adams     5346    11/4/63    28765
Sally    Chang     1654    7/22/54    650000
Billy    Black     1683    9/23/44    336500

% nawk '{print NR, $1, $2, $5}'  employees
1 Tom Jones 543354
2 Mary Adams 28765
3 Sally Chang 650000
4 Billy Black 336500
```

说明

打印文件中每一行的记录号(NR)和第 1、第 2、第 5 字段。

范例 6-14

```
% nawk  '{print $0, NF}'  employees
Tom Jones       4444     5/12/66     543354     5
Mary Adams      5346     11/4/63     28765      5
Sally Chang     1654     7/22/54     650000     5
Billy Black     1683     9/23/44     336500     5
```

说明

打印文件中每条记录，后面跟上该记录的字段数。

6.6.3 字段分隔符

输入字段分隔符 awk 的内置变量 FS 中保存了输入字段分隔符的值。使用 FS 的默认值时，awk 用空格或制表符来分隔字段，并且删除各字段前多余的空格或制表符。可以通过在 BEGIN 语句中或命令行上赋值来改变 FS 的值。接下来我们就要在命令行上给 FS 指定



一个新的值。在命令行上改变 FS 的值需要使用-F 选项，后面指定代表新分隔符的字符。

从命令行改变字段分隔符 范例 6-15 演示了如何使用-F 选项在命令行中改变输入字段分隔符。

范例 6-15

```
% cat employees
Tom Jones:4424:5/12/66:543354
Mary Adams:5346:11/4/63:28765
Sally Chang:1654:7/22/54:650000
Billy Black:1683:9/23/44:336500

% nawk -F: '/Tom Jones/{print $1, $2}'  employees2
Tom Jones  4424
```

说明

-F 选项用来在命令行重新设置输入字段分隔符的值。当冒号紧跟在-F 选项的后面时，nawk 就会在文件中查找冒号，用以分隔字段。

使用多个字段分隔符 你可以指定多个输入字段分隔符。如果有多个字符被用于字段分隔符 FS，则 FS 对应是一个正则表达式字符串，并且被括在方括号中。下面的范例中，字段分隔符是空格、冒号或制表符(旧版的 awk 不支持这一功能)。

范例 6-16

```
% nawk -F'[ :\t]'  '{print $1, $2, $3}' employees
Tom Jones 4424
Mary Adams 5346
Sally Chang 1654
Billy Black 1683
```

说明

-F 选项后面跟了一个位于方括号中的正则表达式。当遇到空格、冒号或制表符时，nawk 会把它当成字段分隔符。这个表达式两头加了引号，这样就不会被 shell 当成自己的元字符来解释(注意，shell 使用方括号来进行文件名扩展)。

输出字段分隔符 默认的输出字段分隔符是单个空格，被保存于 awk 的内置变量 OFS 中。此前的所有例子中，我们都是用 print 语句把输出打印到屏幕上。因此，无论 OFS 如何设置，print 语句中用于分隔字段的逗号，在输出时都被转换成 OFS 的值。如果用 OFS 的默认值，则$1 和$2 之间的逗号会被转换为单个空格，print 函数打印这两个字段时会在它们之间加一个空格。

如果没有用逗号来分隔字段，则输出结果中的字段将堆在一起，除非字段间有逗号分隔，否则输出结果的字段间不会加上 OFS 的值。另外，OFS 的值可以改变。

范例 6-17

```
% cat employees2
Tom Jones:4424:5/12/66:543354
Mary Adams:5346:11/4/63:28765
Sally Chang:1654:7/22/54:650000
```

```
Billy Black:1683:9/23/44:336500
(命令行)
% nawk -F: '/Tom Jones/{print $1, $2, $3, $4}' employees2
Tom Jones  4424 5/12/66  543354
```

说明

输出字段分隔符，即空格，保存在 nawk 的变量 OFS 中。字段间的逗号在输出时被转换成 OFS 中的值。字段被打印到标准输出上，字段之间用空格分隔。

范例 6-18

```
% nawk -F: '/Tom Jones/{print $0}' employees2
Tom Jones:4424:5/12/66:543354
```

说明

变量$0 按输入文件中的原样保存当前记录。记录被原封不动地显示到屏幕上。

6.7 模式与操作

6.7.1 模式

awk 模式用来控制 awk 对输入的文本行执行什么操作。模式由正则表达式、判别条件真伪的表达式或二者的组合构成。awk 的默认操作是打印所有使表达式结果为真的文本行。模式表达式中暗含着 if 语句。如果模式表达式含有 if(如果)的意思，就不必用花括号把它括起来。当 if 是显式地给出时，这个表达式就成了操作语句，语法将不一样(参见 6.17 节“条件语句”)。

范例 6-19

```
% cat employees
Tom Jones       4424      5/12/66     543354
Mary Adams      5346      11/4/63     28765
Sally Chang     1654      7/22/54     650000
Billy Black     1683      9/23/44     336500

(命令行)
1   nawk '/Tom/' employees
Tom Jones      4424      5/12/66     543354

2   nawk '$3 < 4000' employees
    Sally Chang     1654      7/22/54     650000
    Billy Black     1683      9/23/44     336500
```

说明

1. 如果在输入文件中匹配到模式 Tom，则打印 Tom 所在的记录。如果没有显式地指定操作，默认操作是打印文本行，等价于命令：




```
nawk '$0 ~ /Tom/{print $0}' employees
```

2. 如果第 3 个字段的值小于 4000，则打印该记录。

6.7.2 操作

操作(action)是花括号中以分号分隔的语句[②]。如果操作前面有个模式，则该模式控制执行操作的时间。操作可以是简单的语句或复杂的语句组。同一行内的多条语句由分号分隔，独占一行的语句则以换行符分隔。

格式

```
{action}
```

范例 6-20

```
{ print $1, $2 }
```

说明

该操作用来打印前两个字段。

模式可以与操作结合使用。记住，操作是括在花括号中的语句。模式控制它后面第一个左花括号到第一个右花括号之间的操作。操作如果紧跟在某个模式后面，它的第一个左花括号就必须与该模式同处一行。模式永远不会出现在花括号中。

格式

```
 模式 { 操作语句; 操作语句; ...; }
或
 模式 {
  操作语句
  操作语句
}
```

范例 6-21

```
% nawk '/Tom/{print "Hello there, " $1}' employees
Hello there, Tom
```

说明

如果记录中包含模式 Tom，就会打印出字符串“Hello there，Tom”。

如果没有为模式指定操作，就会打印所有匹配该模式的文本行。用于匹配字符串的模式包括了夹在两个正斜杠之间的正则表达式。

---

② 某些版本的 awk 要求必须用分号或换行符来分隔操作，而且花括号中的语句也必须用分号或换行符来分隔。SVR4 的 awk 就要求用分号或换行符分隔同一操作中的多条语句，但是多个操作之间则不要求用分号进行分隔，例如下面这个例子中两个操作之间的分号就可以省掉：

nawk '/Tom/{printf"hi Tom"};{x=5}' file

6.8 正则表达式

对 awk 而言，正则表达式是置于两个正斜杠之间、由字符组成的模式。awk 支持使用(与 egrep 相同的)正则表达式元字符对正则表达式进行某种方式的修改。如果输入行中的某个字符串与正则表达式相匹配，则最终条件为真，于是执行与该表达式关联的所有操作。如果没有指定操作，则打印与正则表达式匹配的记录。参见表 6-5。

范例 6-22

```
% nawk '/Mary/' employees
Mary Adams    5346    11/4/63    28765
```

说明

显示文件 employees 中所有包含正则表达式模式 Mary 的行。

范例 6-23

```
% nawk '/Mary/{print $1, $2}' employees
Mary Adams
```

说明

显示文件 employees 中所有包含正则表达式模式 Mary 的行的头两个字段。

表 6-5 awk 的正则表达式元字符

元 字 符	说 明
^	在串首匹配
$	在串尾匹配
.	匹配单个任意字符
*	匹配零个或多个前导字符
+	匹配一个或多个前导字符
?	匹配零个或一个前导字符
[ABC]	匹配指定字符组(即 A、B 和 C)中任一字符
[^ABC]	匹配任何一个不在指定字符组(即 A、B 和 C)中的字符
[A-Z]	匹配 A 至 Z 之间的任一字符
A|B	匹配 A 或 B
(AB)+	匹配一个或多个 AB 的组合，例如：AB、ABAB、ABABAB
*	匹配星号本身
&	用在替代串中，代表查找串中匹配到的内容

大多数版本的 grep 和 sed 都支持表 6-6 中列出的元字符，但是任何版本的 awk 都不支持。



表 6-6 awk 不支持的元字符

元 字 符	说 明
\\<\\>	单词定位
\\(\\)	向前引用
\\{\\}	重复

6.8.1 匹配整行

如果没有指定操作，则单个正则表达式将对整行进行模式匹配，并打印出所匹配的行。可以使用元字符^来表示需要进行行首匹配的正则表达式。

范例 6-24

```
% nawk '/^Mary/' employees
Mary Adams     5346     11/4/63     28765
```

说明

显示文件 employees 中所有以正则表达式 Mary 开头的行。

范例 6-25

```
% nawk '/^[A-Z][a-z]+ /' employees
Tom Jones       4424     5/12/66     543354
Mary Adams      5346     11/4/63     28765
Sally Chang     1654     7/22/54     650000
Billy Black     1683     9/23/44     336500
```

说明

显示文件 employees 中所有以大写字母开头、后跟一个或多个小写字母、再跟一个空格的行。

6.8.2 匹配操作符

匹配操作符(~)用于对记录或字段的表达式进行匹配。

范例 6-26

```
% cat employees
Tom Jones       4424      5/12/66      543354
Mary Adams      5346      11/4/63      28765
Sally Chang     1654      7/22/54      650000
Billy Black     1683      9/23/44      336500

% nawk '$1 ~ /[Bb]ill/' employees
Billy Black    1683    9/23/44    336500
```

说明

显示所有在第一个字段里匹配到 Bill 或 bill 的行。

范例 6-27

```
% nawk '$1 !~ /ly$/' employees
Tom Jones       4424     5/12/66     543354
Mary Adams      5346     11/4/63     28765
```

说明

显示所有第一个字段不是以 ly 结尾的行。

POSIX 字符类 POSIX(the Portable Operating System Interface，可移植操作系统接口)是一种工业标准，确保程序可以跨操作系统移植。为了保证可移植，POSIX 可以识别字符、阿拉伯数字和符号在不同国家或不同场合的编码方法，以及时间和时期的不同表示。为了处理不同类型的字符，POSIX 增加了基本的和扩展的正则表达式，表 6-7 对括号字符类进行了说明。gawk 支持这些新的元字符类，而 awk 和 nawk 不支持。

表 6-7 POSIX 增加的括号字符类

括 号 类	含 义
[:alnum:]	字母数字字符
[:alpha:]	字母字符
[:cntrl:]	控制字符
[:digit:]	数字字符
[:graph:]	非空白字符(非空格、控制字符等)
[:lower:]	小写字母
[:print:]	与[:graph:]相似，但是包含空格字符
[:punct:]	标点字符
[:space:]	所有的空白字符(换行符、空格、制表符)
[:upper:]	大写字母
[:xdigit:]	允许十六进制的数字(0-9a-fA-F)

在类中，[:alnum:]是另一种表示 A-Z、a-z 和 0-9 的形式，使用这种类时，必须要用另外一个方括号扩起来，例如“A-Za-z0-9”自己本身不是正则表达式，而[A-Za-z0-9]则是正则表达式。类似地，[:alnum:] 应该写为[[:alnum:]]。第一种形式[A-Za-z0-9]与方括号形式[[:alnum:]]的不同之处在于，前者依赖于 ASCII 字符编码的形式，而第二种形式允许其他语言的字符在类中表示，例如瑞典语和德语。

范例 6-28

```
% gawk '/[[:lower:]]+g[[:space:]]+[[:digit:]]/' employees
Sally Chang 1654 7/22/54 650000
```

说明

gwak 搜索一个或多个小写字母，后面跟着一个字母“g”，再后面为一个或多个空格，然后是一个数字的模式(如果是在 Linux 上使用的 awk，则应当明白 awk 是链接到 gawk 上



的，也就是说 awk 和 gawk 的命令同样有效)。

6.9 脚本文件中的 awk 命令

如果有多条 awk 的模式/操作语句要处理，把它们写在脚本里通常会方便得多。脚本是一个包含 awk 注释和语句的文件。如果同一行中有多条语句或操作，必须用分号将它们隔开。如果每条语句都在不同的行上，就不需要用分号来分隔。如果操作跟在某个模式后面，它的左花括号就必须与该模式位于同一行。注释要以井号(#)开头。

范例 6-29

```
% cat employees
Tom Jones:4424:5/12/66:54335
Mary Adams:5346:11/4/63:28765
Billy Black:1683:9/23/44:336500
Sally Chang:1654:7/22/54:65000
Jose Tomas:1683:9/23/44:33650

(awk 脚本)
% cat info
1   # My first awk script by Jack Sprat
    # Script name: info; Date: February 28, 2004
2   /Tom/{print  "Tom's birthday is "$3}
3   /Mary/{print NR, $0}
4   /^Sally/{print "Hi Sally. "  $1 " has a salary of  $" $4 "."}
    # End of info script

(命令行)
5   % nawk -F: -f info employees2
    Tom's birthday is 5/12/66
    2 Mary Adams:5346:11/4/63:28765
    Hi Sally. Sally Chang has a salary of $65000.
```

说明

1. 这两行是注释。

2. 如果在输入的文本行中匹配到正则表达式 Tom，则打印字符串“Tom's birthday is”和第三个字段($3)的值。

3. 如果在输入的文本行中匹配到正则表达式 Mary，则该操作块将打印当前记录的编号 NR 和内容。

4. 如果在输入文本行的开头找到正则表达式 Sally，则依次打印字符串“Hi Sally.”、第一个字段($1)的值、字符串“has a salary of $”和第 4 个字段($4)的值。

5. nawk 命令后面跟着-F: 选项，用于将字段分隔符指定为冒号。-f 选项后跟的是 awk 脚本的名称。awk 将从文件 info 中读取指令。最后那个参数 employee2 是输入文件的名称。

6.10 复习

这一节中的范例将用到一个名为 datafile 的示例数据库，为方便您查阅它将重复出现。在这个数据库中：输入字段分隔符 FS 是默认的空白符。字段数 NF 是 8，字段数因行而异，但该文件中则是固定的。记录分隔符 RS 是换行符，用于分隔文件中的各行。awk 把各条记录的编号保存在变量 NR 中。输出字段分隔符 OFS 是空格，虽然输入行用逗号来分隔字段，打印输出的字段之间却是用空格来分隔。

```
% cat datafile
northwest    NW    Joel Craig       3.0    .98    3    4
western      WE    Sharon Kelly     5.3    .97    5    23
southwest    SW    Chris Foster     2.7    .8     2    18
southern     SO    May Chin         5.1    .95    4    15
southeast    SE    Derek Johnson    4.0    .7     4    17
eastern      EA    Susan Beal       4.4    .84    5    20
northeast    NE    TJ Nichols       5.1    .94    3    13
north        NO    Val Shultz       4.5    .89    5    9
central      CT    Sheri Watson     5.7    .94    5    13
```

6.10.1 简单的模式匹配

范例 6-30

```
nawk '/west/' datafile
northwest          NW      Joel Craig          3.0   .98    3    4
western            WE      Sharon Kelly        5.3   .97    5    23
southwest          SW      Chris Foster        2.7   .8     2    18
```

说明

打印所有包含模式 west 的行。

范例 6-31

```
nawk '/^north/' datafile
northwest           NW     Joel Craig          3.0   .98    3    4
northeast           NE     TJ Nichols          5.1   .94    3    13
north               NO     Val Shultz          4.5   .89    5    9
```

说明

打印所有以模式 north 开头的行。

范例 6-32

```
nawk '/^(no|so)/' datafile
northwest          NW     Joel Craig          3.0    .98    3    4
```



```
southwest       SW    Chris Foster     2.7   .8    2   18
southern        SO    May Chin         5.1   .95   4   15
southeast       SE    Derek Johnson    4.0   .7    4   17
northeast       NE    TJ Nichols       5.1   .94   3   13
north           NO    Val Shultz       4.5   .89   5   9
```

说明

打印所有以模式 no 或 so 开头的行。

6.10.2 简单的操作

范例 6-33

```
nawk '{print $3, $2}' datafile
Joel  NW
Sharon  WE
Chris  SW
May  SO
Derek  SE
Susan  EA
TJ  NE
Val  NO
Sheri  CT
```

说明

输出字段分隔符 OFS 默认是空格。$3 和$2 之间的逗号被转换为 OFS 的值。打印的结果是第 3 个字段，后面跟一个空格和第 2 个字段。

% cat datafile

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17
eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

范例 6-34

```
nawk '{print $3 $2}' datafile
JoelNW
SharonWE
ChrisSW
MaySO
DerekSE
SusanEA
```

```
TJNE
ValNO
SheriCT
```

说明

第 2 个字段跟在第 3 个字段后面。由于$3 和$2 之间没有用逗号分隔，所以显示的输出中字段之间没有空格。

范例 6-35

```
nawk 'print $1' datafile
nawk: syntax error at source line 1
context is
        >>> print <<<  $1
nawk: bailing out at source line 1
```

说明

输出的结果是 nawk(新版 awk)的报错信息。nawk 的报错信息比旧 awk 的详细得多。这个例子中，操作语句中漏掉了花括号。

范例 6-36

```
awk 'print $1' datafile
awk: syntax error near line 1
awk: bailing out near line 1
```

说明

这是 awk(旧版 awk)的报错信息。旧版的 awk 程序很难调试，因为几乎所有的报错信息都一样。这个例子的错误是操作语句中缺少了花括号。

范例 6-37

```
nawk '{print $0}' datafile
northwest      NW    Joel Craig       3.0   .98   3   4
western        WE    Sharon Kelly     5.3   .97   5   23
southwest      SW    Chris Foster     2.7   .8    2   18
southern       SO    May Chin         5.1   .95   4   15
southeast      SE    Derek Johnson    4.0   .7    4   17
eastern        EA    Susan Beal       4.4   .84   5   20
northeast      NE    TJ Nichols       5.1   .94   3   13
north          NO    Val Shultz       4.5   .89   5   9
central        CT    Sheri Watson     5.7   .94   5   13
```

说明

打印所有记录。$0 保存的是当前记录。

范例 6-38

```
nawk '{print "Number of fields: "NF}' datafile
Number of fields: 8
Number of fields: 8
Number of fields: 8
```

```
Number of fields: 8
Number of fields: 8
Number of fields: 8
Number of fields: 8
Number of fields: 8
Number of fields: 8
```

说明

每个记录都有 8 个字段。awk 的内置变量 NF 用来保存记录的字段数，在处理每条记录时都会被重置。

% cat datafile

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17
eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

6.10.3 模式与操作组合的正则表达式

范例 6-39

```
nawk '/northeast/{print $3, $2}' datafile
TJ NE
```

说明

如果记录包含(或匹配)模式 northest，则打印它的第 3 个字段，后跟空格和第 2 个字段。

范例 6-40

```
nawk '/E/' datafile
western          WE     Sharon Kelly        5.3  .97  5   23
southeast        SE     Derek Johnson       4.0  .7   4   17
eastern          EA     Susan Beal          4.4  .84  5   20
northeast        NE     TJ Nichols          5.1  .94  3   13
```

说明

如果记录中含有字母 E，就打印整条记录。

范例 6-41

```
nawk '/^[ns]/{print $1}' datafile
northwest
southwest
```

```
southern
southeast
northeast
north
```

说明

如果记录以 n 或 s 开头，就打印第 1 个字段。

范例 6-42

```
nawk '$5 ~ /\.[7-9]+/' datafile
southwest        SW     Chris Foster      2.7   .8    2    18
central          CT     Sheri Watson      5.7   .94   5    13
```

说明

如果某条记录的第 5 个字段包含一个句点，而且句点后是一或多个 7~9 之间的数字，就打印该记录。

范例 6-43

```
nawk '$2 !~ /E/{print $1, $2}' datafile
northwest NW
southwest SW
southern SO
north NO
central CT
```

说明

如果某条记录的第 2 个字段不含模式 E，则打印该记录的第一个字段，隔一个空格再打印第 2 个字段($1,$2)。

范例 6-44

```
nawk '$3 ~ /^Joel/{print $3 " is a nice guy."}' datafile
Joel is a nice guy.
```

说明

如果第 3 个字段以模式 Joel 开头，则打印该字段，并且在后面跟上字符串 " is a nice guy."。注意，如果要打印空格，就要把它包括在字符串中。

例 6-45

```
nawk '$8 ~ /[0-9][0-9]$/{print $8}' datafile
23
18
15
17
20
13
13
```

说明

如果第 8 个字段以两个数字结尾，则打印该字段。



```
% cat datafile
northwest    NW    Joel Craig       3.0    .98    3    4
western      WE    Sharon Kelly     5.3    .97    5    23
southwest    SW    Chris Foster     2.7    .8     2    18
southern     SO    May Chin         5.1    .95    4    15
southeast    SE    Derek Johnson    4.0    .7     4    17
eastern      EA    Susan Beal       4.4    .84    5    20
northeast    NE    TJ Nichols       5.1    .94    3    13
north        NO    Val Shultz       4.5    .89    5    9
central      CT    Sheri Watson     5.7    .94    5    13
```

范例 6-46

```
nawk '$4 ~ /Chin$/{print "The price is $" $8 "."}' datafile
The price is $15.
```

说明

如果第 4 个字段以 Chin 结尾，则打印双引号间的字符串(“The price is $”)、第 8 个字段($8)和最后那个只含一个句点的字符串。

范例 6-47

```
nawk '/TJ/{print $0}' datafile
northeast          NE       TJ Nichols          5.1    .94   3   13
```

说明

如果记录中包含模式 TJ，则打印该记录($0)。

6.10.4 输入字段分隔符

范例 6-48 到范例 6-52 都将使用下面的 datafile2 文件。

```
% cat datafile2
Joel Craig:northwest:NW:3.0:.98:3:4
Sharon Kelly:western:WE:5.3:.97:5:23
Chris Foster:southwest:SW:2.7:.8:2:18
May Chin:southern:SO:5.1:.95:4:15
Derek Johnson:southeast:SE:4.0:.7:4:17
Susan Beal:eastern:EA:4.4:.84:5:20
TJ Nichols:northeast:NE:5.1:.94:3:13
Val Shultz:north:NO:4.5:.89:5:9
Sheri Watson:central:CT:5.7:.94:5:13
```

范例 6-48

```
nawk '{print $1}' datafile2
```

```
Joel
Sharon
Chris
May
Derek
Susan
TJ
Val
Sheri
```

说明

默认的输入字段分隔符是空白符。本例打印第 1 个字段($1)。

范例 6-49

```
nawk -F: '{print $1}' datafile2
Joel Craig
Sharon Kelly
Chris Foster
   <more output here>
Val Shultz
Sheri Watson
```

说明

-F 选项指定以冒号作为输入字段分隔符。本例打印第一个字段($1)。

范例 6-50

```
nawk '{print "Number of fields: "NF}' datafile2
Number of fields: 2
Number of fields: 2
Number of fields: 2
   <more of the same output here>
Number of fields: 2
Number of fields: 2
```

说明

由于字段分隔符还是默认值(空白符)，所以每条记录的字段数都是 2。记录中唯一的空格位于名和姓之间。

范例 6-51

```
nawk -F: '{print "Number of fields: "NF}' datafile2
Number of fields: 7
Number of fields: 7
Number of fields: 7
   <more of the same output here>
Number of fields: 7
Number of fields: 7
```

说明

因为字段分隔符被设为冒号，所以每条记录的字段数变成了 7。



范例 6-52

```
nawk -F"[ :]" '{print $1, $2}' datafile2
Joel Craig northwest
Sharon Kelly western
Chris Foster southwest
May Chin southern
Derek Johnson southeast
Susan Beal eastern
TJ Nichols northeast
Val Shultz north
Sheri Watson central
```

说明

使用 nawk 命令，可以用正则表达式来指定多个字段分隔符。空格或冒号将被指定为是字段分隔符。本例打印头两个字段(方括号必须在引号内，以防止 shell 将它们解释成 shell 元字符)。

6.10.5 编写 awk 脚本

本节范例使用了下面的 datafile 文件。

% cat datafile

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17
eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

范例 6-53

```
cat nawk.sc1
# This is a comment
# This is my first nawk script
1    /^north/{print $1, $2, $3}
2    /^south/{print "The " $1 " district."}

3    nawk -f nawk.sc1 datafile
     northwest NW Joel
     The southwest district.
     The southern district.
     The southeast district.
     northeast NE TJ
     north NO Val
```

说明

1. 如果记录以模式 north 开头，则打印头 3 个字段($1,$2,$3)。

2. 如果记录以模式 south 开头，则依次打印字符串 The，第一个字段的值($1)和字符串 district。

3. –f 选项后面跟的是 awk 脚本文件名，接下来的那个参数是要处理的输入文件名。

习题 3 awk 练习

(参见从本书合作站点下载的文件中名为 lab3.data 的数据库)

Mike Harrington:(510) 548-1278:250:100:175
Christian Dobbins:(408) 538-2358:155:90:201
Susan Dalsass:(206) 654-6279:250:60:50
Archie McNichol:(206) 548-1348:250:100:175
Jody Savage:(206) 548-1278:15:188:150
Guy Quigley:(916) 343-6410:250:100:175
Dan Savage:(406) 298-7744:450:300:275
Nancy McNeil:(206) 548-1278:250:80:75
John Goldenrod:(916) 348-4278:250:100:175
Chet Main:(510) 548-5258:50:95:135
Tom Savage:(408) 926-3456:250:168:200
Elizabeth Stachelin:(916) 440-1763:175:75:300

该数据库包含姓名、电话号码、以及在最近 3 个月中的竞选捐款数额。

1. 打印所有的电话号码。
2. 打印 Dan 的电话号码。
3. 打印 Susan 的姓名和电话号。
4. 打印所有性以 D 开头的姓氏。
5. 打印所有以 C 或 E 开头的名字。
6. 打印所有只包含 4 个字符的名字。
7. 打印所有区号为 916 的人的名字。
8. 打印 Mike 的资助金额，每一个值要使用美元符开头，例如“$250 $100 $175”。
9. 打印所有的姓，后面跟着一个逗号和名。
10. 编写一个名为 facts 的脚本，并完成下面的工作：
    a. 打印所有姓氏为 Savage 的人的全名和电话号码。
    b. 打印 Chet 的资助金额。
    c. 打印所有在第一个月资助了 250 美元的人。

6.11 比较表达式

比较表达式用来对文本行进行比较，只有条件为真，才执行指定的动作。比较表达式使用关系运算符，用于比较数字与字符串。



6.11.1 关系运算符

表 6-8 列出了所有关系运算符。关系表达式的计算结果为真时，表达式的值为 1；反之，则为 0。

表 6-8 关系运算符

运 算 符	含 义	示 例
<	小于	x < y
<=	小于或等于	x <= y
==	等于	x == y
!=	不等于	x != y
>=	大于或等于	x >= y
>	大于	x > y
~	与正则表达式匹配	x ~ /y/
!~	与正则表达式不匹配	x !~ /y/

范例 6-54

(数据库)
```
% cat employees
    Tom Jones        4423      5/12/66           543354
    Mary Adams       5346      11/4/63           28765
    Sally Chang      1654      7/22/54           650000
    Billy Black      1683      9/23/44           336500
```

(命令行)
```
1   % awk '$3 == 5346' employees
    Mary Adams       5346       11/4/63           28765

2   % awk '$3 > 5000{print $1} ' employees
    Mary

3   % awk '$2 ~ /Adam/ ' employees
    Mary Adams       5346       11/4/63           28765

4   % awk '$2 !~ /Adam/ ' employees
    Tom Jones        4423       5/12/66           543354
    Sally Chang      1654       7/22/54           650000
    Billy Black      1683       9/23/44           336500
```

说明

1. 如果某行的第 3 个字段的值等于 5346，则条件为真，awk 将执行默认操作——打印该行。如果表达式中隐含着 if 条件，它就是一个条件模式测试。

2. 如果某条记录的第 3 个字段的值大于 5000，awk 就打印该记录的第 1 个字段。

3. 如果记录的第 2 个字段匹配正则表达式 Adam，则打印该记录。

4. 如果记录的第 2 个字段不匹配正则表达式 Adam，则打印该记录。假设表达式的值是一个数，与其比较的却是个字符串值，二者之间的运算符如果是用于数值比较的运算符，则字符串值将被转换为一个数值，如果是用于字符串比较的运算符，则数值被转换为字符串值。

6.11.2 条件表达式

条件表达式的运算要用到两个符号：问号和冒号。条件表达式其实就是 if/else 语句的捷径，它们有着相同的结果。条件表达式的一般格式如下所示。

格式

条件表达式 1 ？ 表达式 2 ： 表达式 3

上面这条语句能够产生与下面的 if/else 语句相同的结果(后面将对 if/else 结构进行全面讨论)。

```
{
if (expression1)
        expression2
else
        expression3
}
```

范例 6-55

```
% awk '{max=($1 > $2) ? $1 : $2; print max}' filename
```

说明

如果记录的第 1 个字段的值大于第 2 个字段的值，则把问号后面那个表达式的值赋给 max，否则就将冒号后面那个表达式的值赋给 max。

这条命令相当于：

```
if ($1 > $2)
      max=$1
else
      max=$2
```

6.11.3 算术运算

可以在模式中执行计算。awk(所有版本的)都将按浮点方式执行算术运算。表 6-9 列出了所有的算术运算符。

范例 6-56

```
% awk '$3 * $4 > 500' filename
```



表 6-9 算术运算符

运 算 符	含 义	例 子
+	加	x +y
-	减	x + y
*	乘	x * y
/	除	x / y
%	模	x % y
^	幂	x ^ y

说明

awk 将记录的第 3 个字段($3)与第 4 个字段($4)的值相乘，如果乘积大于 500，则显示该行(假定 filename 是含有输入数据的文件)。

6.11.4 逻辑操作符和复合模式

逻辑操作符用来测试表达式或者模式的真假。符号&&，表示逻辑与，当所有表达式均为真时，整个表达式为真，只要有一个表达式为假，整个表达式就为假。符号||，表示逻辑或，只要有一个表达式或者模式为真，整个表达式就为真。如果所有表达式为假，则整个表达式为假。

复合模式是用逻辑运算符(参见表 6-10)将模式组合起来形成的表达式。表达式的计算是从左往右的。

表 6-10 逻辑运算符

运 算 符	含 义	例 子
&&	逻辑与	a && b
||	逻辑或	a || b
!	逻辑非	!a

范例 6-57

```
% awk '$2 > 5 && $2 <= 15' filename
```

说明

awk 将显示同时符合这两个条件的行，即该行的第 2 个字段($2)的值大于 5，且小于或等于 15。运算符&&要求两个条件都必须为真(假定 filename 是包含输入数据的文件)。

范例 6-58

```
% awk '$3 == 100 || $4 > 50' filename
```

说明

awk 将显示符合两个条件之一的行，即第 3 个字段等于 100 或第 4 个字段大于 50 的行。运算符||只要求有一个条件必须为真(假定 filename 是包含输入数据的文件)。

范例 6-59

```
% awk '!($2 < 100 && $3 < 20)' filename
```

说明

如果两个条件都为真，awk 将否定该表达式并取消对该行的打印。所以显示出来的行都有一个条件为假，甚至两个条件都为假。一元运算符！对条件的结果求反，所以，如果表达式本来产生一个为真的条件，“非”操作会将其变为假，反之亦然(假定 filename 是包含输入数据的文件)。

6.11.5 范围模式

范围模式先匹配从第一个模式的首次出现到第二个模式的首次出现之间的内容，然后匹配从第一个模式的下一次出现到第二个的下一次出现，以此类推。如果匹配到第一个模式而没有发现第二个模式，awk 就将显示从第一个模式首次出现的行到文件末尾之间的所有行。

范例 6-60

```
% awk '/Tom/,/Suzanne/' filename
```

说明

awk 将显示从 Tom 首次出现的行到 Suzanne 首次出现的行这个范围内的所有行，包括两个边界在内。如果没有找到 Suzanne，awk 将继续打印各行直至文件末尾。如果打印完 Tom 到 Suzanne 的内容之后，又出现了 Tom，awk 就又一次开始显示行，直至找到下一个 Suzanne 或文件末尾。

6.11.6 验证数据合法性

下面这个口令核对程序出自 *The AWK Programming Language*[③]一书，它所用的都是我们目前已经讨论过的 awk 命令。我们通过这个程序来说明如何验证文件中的数据。

范例 6-61

```
( Password 数据库)
1 % cat /etc/passwd
tooth:pwHfudo.eC9sM:476:40:Contract
Admin.:/home/rickenbacker/tooth:/bin/csh
lisam:9JY7OuS2f3lHY:4467:40:Lisa M.
Spencer:/home/fortune1/lisam:/bin/csh
goode:v7Ww.nWJCeSIQ:32555:60:Goodwill Guest User:/usr/goodwill:/bin/csh
bonzo:eTZbu6M2jM7VA:5101:911: SSTOOL Log
account :/home/sun4/bonzo:/bin/csh
info:mKZsrioPtW9hA:611:41:Terri Stern:/home/chewie/info:/bin/csh
```

③ *The AWK Programming Language* 的作者是 Aho、Weinberger 和 Kernighan，由波士顿的 Addison-Wesley 公司于 1988 出版。



```
    cnc:IN1IVqVj1bVv2:10209:41:Charles Carnell:/home/christine/cnc:/bin/csh
    bee:*:347:40:Contract Temp.:/home/chanel5/bee:/bin/csh
    friedman:oyuIiKoFTV0TE:3561:50:Jay
Friedman:/home/ibanez/friedman:/bin/csh
    chambers:Rw7R1k77yUY4.:592:40:Carol
Chambers:/usr/callisto2/chambers:/bin/csh
    gregc:nkLulOg:7777:30:Greg Champlin FE Chicago
    ramona:gbDQLdDBeRc46:16660:68:RamonaLeininge MWA CustomerService
Rep:/home/forsh:
```

```
(Awk 命令)
2   % cat /etc/passwd | awk -F: '\
3   NF != 7{\
4   printf("line %d, does not have 7 fields: %s\n",NR,$0)} \
5   $1 !~ /[A-Za-z0-9]/{printf("line %d, nonalphanumeric user id:
    %s\n",NR,$0)} \
6   $2 == "*" {printf("line %d, no password: %s\n",NR,$0)} '
```

```
(输出)
        line 7, no password: bee:*:347:40:Contract
Temp.:/home/chanel5/bee:/bin/csh
        line 10, does not have 7 fields: gregc:nk2EYi7kLulOg:7777:30:Greg
Champlin FE Chicago
        line 11, does not have 7 fields: ramona:gbDQLdDBeRc46:16660:68:Ramona
        Leininger MWA Customer Service Rep:/home/forsh:
```

说明

1. 显示文件/etc/passwd 的内容。

2. cat 程序把输出送给 awk。awk 的字段分隔符是冒号。

3. 如果字段数(NF)不等于 7，则执行接下来的操作块。

4. printf 函数打印字符串“line <行号>, does not have 7 fields:”，后面跟上当前记录的记录号(NR)和记录本身($0)。

5. 如果第 1 个字段($1)中不含任何字母和数字字符，printf 函数就打印字符串“nonalphanumberic user id:”，后面跟上当前记录的记录号和内容。

6. 如果第 2 个字段($2)是一个星号，就打印字符串“no passwd:”，后面跟上记录号和记录本身。

6.12 复习

本节示例使用了下面的 datafile 样本数据库，为了方便查阅，我们再次给出这个样本数据库。

```
% cat datafile
northwest    NW    Joel Craig       3.0    .98    3    4
western      WE    Sharon Kelly     5.3    .97    5    23
southwest    SW    Chris Foster     2.7    .8     2    18
southern     SO    May Chin         5.1    .95    4    15
southeast    SE    Derek Johnson    4.0    .7     4    17
eastern      EA    Susan Beal       4.4    .84    5    20
northeast    NE    TJ Nichols       5.1    .94    3    13
north        NO    Val Shultz       4.5    .89    5    9
central      CT    Sheri Watson     5.7    .94    5    13
```

6.12.1 相等性测试

范例 6-62

```
% awk '$7 == 5' datafile
western         WE    Sharon Kelly     5.3   .97   5   23
eastern         EA    Susan Beal       4.4   .84   5   20
north           NO    Val Shultz       4.5   .89   5   9
central         CT    Sheri Watson     5.7   .94   5   13
```

说明

如果记录的第 7 个字段($7)等于数字 5，就打印该记录。

范例 6-63

```
% awk '$2 == "CT"{print $1, $2}' datafile
central        CT
```

说明

如果记录的第 2 个字段($2)等于字符串“CT”，就打印它的第 1 和第 2 个字段($1, $2)。字符串必须加引号。

范例 6-64

```
% awk '$7 != 5' datafile
northwest       NW    Joel Craig      3.0   .98   3   4
southwest       SW    Chris Foster    2.7   .8    2   18
southern        SO    May Chin        5.1   .95   4   15
southeast       SE    Derek Johnson   4.0   .7    4   17
northeast       NE    TJ Nichols      5.1   .94   3   13
```

说明

如果记录的第 7 个字段不等于数字 5，就打印该记录。

6.12.2 关系运算符

范例 6-65

```
% awk '$7 < 5 {print $4, $7}' datafile
Craig 3
Foster 2
Chin 4
Johnson 4
Nichols 3
```

说明

如果记录的第 7 个字段的值小于 5，就打印它的第 4 和第 7 个字段。

范例 6-66

```
% awk '$6 > .9 {print $1, $6}' datafile
northwest .98
western .97
southern .95
northeast .94
central .94
```

说明

如果记录的第 6 个字段大于.9，就打印它的第 1 和第 6 个字段。

范例 6-67

```
% awk '$8 <= 17 { print $8}' datafile
4
15
17
13
9
13
```

说明

如果记录的第 8 个字段小于或等于 17，就打印它。

% cat datafile

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17
eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

范例 6-68

```
% awk '$8 >= 17 {print $8}' datafile
23
18
17
20
```

说明

如果记录的第 8 个字段($8)大于或等于 17，则打印它。

6.12.3 逻辑运算符

范例 6-69

```
% awk '$8 > 10 && $8 < 17' datafile
southern        SO      May Chin        5.1    .95    4    15
northeast       NE      TJ Nichols      5.1    .94    3    13
central         CT      Sheri Watson    5.7    .94    5    13
```

说明

如果记录的第 8 个字段($8)大于 10 而又小于 17，则打印该记录。只有两个表达式都为真时，记录才会被打印。

范例 6-70

```
% awk '$2 == "NW" || $1 ~ /south/{print $1, $2}' datafile
northwest NW
southwest SW
southern SO
southeast SE
```

说明

如果第 2 个字段($2)等于字符串“NW”或第一个字段($1)包含模式 south，则打印第 1、第 2 字段($1，$2)。只要有一个表达式为真，就执行打印操作。

6.12.4 逻辑非运算符

范例 6-71

```
% awk '!($8 == 13){print $8}' datafile
4
23
18
15
17
20
9
```



说明

如果第 8 个字段($8)等于 13，!(非运算符)就对表达式取反，并取消打印操作。!是一元否定操作符。

6.12.5 算术运算符

范例 6-72

```
% awk '/southern/{print $5 + 10}' datafile
15.1
```

说明

如果记录中包含正则表达式 southern，就将第 5 个字段的值($5)与 10 相加，并打印结果。注意，结果以浮点数形式显示。

范例 6-73

```
% awk '/southern/{print $8 + 10}' datafile
25
```

说明

如果记录中包含正则表达式 southern，就将第 8 个字段的值($8)与 10 相加，并打印结果。注意，结果以整数形式显示。

% cat datafile

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17
eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

范例 6-74

```
% awk '/southern/{print $5 + 10.56}' datafile
15.66
```

说明

如果记录中包含正则表达式 southern，就将 10.56 与第 5 个字段($5)的值相加并显示计算结果。

范例 6-75

```
% awk '/southern/{print $8 - 10}' datafile
```

```
5
```

说明

如果记录中包含正则表达式 southern，就用第 8 个字段($8)的值减去 10，并是显示计算结果。

范例 6-76

```
% awk '/southern/{print $8 / 2}' datafile
7.5
```

说明

如果记录中包含正则表达式 southern，就用第 8 个字段($8)的值除以 2，并显示计算结果。

范例 6-77

```
% awk '/northeast/{print $8 / 3}' datafile
4.33333
```

说明

如果记录中包含正则表达式 northeast，就用第 8 个字段($8)的值除以 3，并显示计算结果。结果精确到小数点后第 5 位。

范例 6-78

```
% awk '/southern/{print $8 * 2}' datafile
30
```

说明

如果记录中包含正则表达式 southern，就用第 8 个字段($8)的值乘以 2，并显示计算结果。

范例 6-79

```
% awk '/northeast/ {print $8 % 3}' datafile
1
```

说明

如果记录中包含正则表达式 northeast，就用第 8 个字段($8)的值除以 3，并显示余数(模数)。

范例 6-80

```
% awk '$3 ~ /^Susan/\
{print "Percentage: "$6 + .2 " Volume: " $8}' datafile
Percentage: 1.04 Volume: 20
```

说明

如果记录的第 3 个字段是以正则表达式 Susan 开头的，print 函数就显示出计算结果和双引号中的字符串。



% cat datafile

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17
eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

6.12.6 范围运算符

范例 6-81

```
% awk '/^western/,/^eastern/' datafile
western         WE      Sharon Kelly     5.3    .97   5    23
southwest       SW      Chris Foster     2.7    .8    2    18
southern        SO      May Chin         5.1    .95   4    15
southeast       SE      Derek Johnson    4.0    .7    4    17
eastern         EA      Susan Beal       4.4    .84   5    20
```

说明

从以正则表达式 western 开头的记录开始，到以正则表达式 eastern 开头的记录结束，这个范围内的所有记录都被打印。如果之后又发现了以 western 开头的记录，就再次打印记录，直至发现以 eastern 开头的记录或到达文件末尾。

6.12.7 条件运算符

范例 6-82

```
% awk '{print ($7 > 4 ? "high "$7 : "low "$7)}' datafile
low 3
high 5
low 2
low 4
low 4
high 5
low 3
high 5
high 5
```

说明

如果第 7 个字段($7)大于 4，print 函数就输出问号后面这个表达式的值(即字符串“high”

和第 7 个字段)，否则，print 函数输出的就是冒号后面那个表达式的值(即字符串“low”和第 7 个字段)。

6.12.8 赋值运算符

范例 6-83

```
% awk '$3 == "Chris"{ $3 = "Christian"; print}' datafile
southwest SW Christian Foster 2.7 .8 2 18
```

说明

如果记录的第 3 个字段($3)等于字符串“Chris”，操作就将“Christian”赋给第 3 个字段，然后显示该记录。双等号检查它的操作数是否相等，而单引号则用于赋值。

范例 6-84

```
% awk '/Derek/{$8 += 12; print $8}' datafile
29
```

说明

如果找到正则表达式 Derek，就将第 8 个字段加上 12，并将结果赋给第 8 个字段并显示。这个运算的另一种书写方式是：$8 = $8 + 12。

范例 6-85

```
% awk '{$7 %= 3; print $7}' datafile
0
2
2
1
1
2
0
2
2
```

说明

对每一条记录执行如下操作：将第 7 个字段除以 3，然后将余数(模数)赋给第 7 个字段并显示。

习题 4 awk 练习

(从本书合作站点下载的文件中有 lab4.data 数据库)

Mike Harrington:(510) 548-1278:250:100:175
Christian Dobbins:(408) 538-2358:155:90:201
Susan Dalsass:(206) 654-6279:250:60:50
Archie McNichol:(206) 548-1348:250:100:175
Jody Savage:(206) 548-1278:15:188:150
Guy Quigley:(916) 343-6410:250:100:175



Dan Savage:(406) 298-7744:450:300:275
Nancy McNeil:(206) 548-1278:250:80:75
John Goldenrod:(916) 348-4278:250:100:175
Chet Main:(510) 548-5258:50:95:135
Tom Savage:(408) 926-3456:250:168:200
Elizabeth Stachelin:(916) 440-1763:175:75:300

上面这个数据库的记录内容包括姓名、电话号码和最近 3 个月的竞选捐款数额。

1. 打印在第二个月捐款超过 100 美元的人的姓和名。
2. 打印在最后一个月捐款少于 85 美元的人的姓名和电话号码。
3. 打印第一个月捐款额在 75~150 美元之间的人。
4. 打印这 3 个月的捐款总额不超过 800 美元的人。
5. 打印月均捐款额大于 200 美元的人的姓名和电话号码。
6. 打印不在 916 区的人的姓。
7. 打印每条记录，并在记录前加上其记录号。
8. 打印每个人的姓名和捐款总额。
9. 把 Chet 第二个月的捐款额加上 10。
10. 把 Nancy McNeil 的名字改成 Louise McInnes。

6.13 变量

6.13.1 数值变量和字符串变量

数值常量可以表示为整数(如 243)、浮点数(如 3.14)或用科学记数法表示的数(如.723E-1 或 3.4e7)。字符串则括在双引号中，例如“Hello world”。

初始化与强制类型转换 只要在 awk 程序中被提到，变量就开始存在。变量可以是一个字符串或一个数字，也可以既是字符串又是数字。变量被设置后，就变成与等号右边那个表达式相同的类型。

未经初始化的变量的值是 0 或" "，究竟是哪个则取决于它们被使用时的上下文。

```
name = "Nancy"   name 是字符串
x++              x 是数字，它被初始化为 0，然后加 1
number = 35      number 是数字
```

如果要将一个字符串强制转换为数字，方法为：

```
name + 0
```

将数字转换成字符串的方法则是：

```
number " "
```

所有由 split 函数创建的字段或数组元素都被视为字符串，除非它们只包含数字值。如果某个字段或数组元素为空，它的值就是空串。空行也可以被视为空串。

6.13.2 用户自定义变量

用户自定义变量的变量名可以由字母、数字和下划线组成，但是不能以数字开头。awk 的变量不用声明其类型，awk 可以从变量在表达式中的上下文推导出它的数据类型。如果变量未被初始化，awk 会将字符串变量初始化为空串，将数值变量初始化为 0。必要时，awk 会将字符型变量转换为数值型变量，或者反向转换。对变量赋值要使用 awk 的赋值运算符。参见表 6-11。

表 6-11 赋值运算符

运 算 符	含 义	等 效 表 达
=	a = 5	a = 5
+=	a = a + 5	a += 5
-=	a = a - 5	a -= 5
*=	a = a * 5	a *= 5
/=	a = a / 5	a /= 5
%=	a =a % 5	a %= 5
^=	a = a ^ 5	a ^= 5

最简单的赋值方式是求出表达式的结果，然后将其赋给变量。

格式

变量=表达式

范例 6-86

```
% nawk '$1 ~ /Tom/ {wage = $2 * $3; print wage}' filename
```

说明

awk 将在第 1 个字段中扫描 Tom。如果发现某一行符合条件，就将其第 2 个字段的值与第 3 个字段的值相乘，乘积赋值给用户定义的变量 wage。由于乘法是算术运算，所以 awk 把 wage 的初始值设为 0(%是 UNIX 的命令提示符，filename 是输入文件的文件名)。

递增和递减运算符　如果要将操作数加 1，可以使用递增运算符。表达式 x++等价于 x = x + 1。类似地，递减运算符则使操作数减少 1。表达式 x--等价于 x = x - 1。当进行循环操作时，如果只需要递增或递减一个计数器，这种运算符就很有用。递增和递减运算符可以放在操作数的前面，如++x；也可以置于操作数之后，如 x++。用于赋值语句时，这两个运算符的位置不同可能会造成运算结果的差异。

```
{ x = 1; y = x++; print x, y }
```



上面这个例子中的++称为后递增运算符：y 先被赋值为 1，然后 x 才加 1。这样，当所有运算做完后，y 等于 1，而 x 等于 2。

```
{ x = 1; y = ++x; print x, y }
```

上面这个例子中的++称为先递增运算符：先将 x 加 1，然后才将值 2 赋给 y。这样，在所有运算完成后，y 等于 2，x 也等于 2。

命令行上的用户自定义变量 可以在命令行上对变量赋值，然后将其传递给 awk 脚本。如果想对参数处理和 ARGV 有更多了解，请参见 6.20.2 节，“处理命令行参数(nawk)”。

范例 6-87

```
nawk -F: -f awkscript   month=4  year=2004 filename
```

说明

用户自定义的变量 month 和 year 分别被赋值为 4 和 2004。在 awk 脚本中可以使用这些变量，就好像它们是在脚本中生成的一样。注意，如果命令行中 filename 的位置在变量之前，这些变量将不能在 BEGIN 语句中使用(参见 6.13.3 节，“BEGIN 模式”)。

(nawk 的)-v 选项 nawk 提供的选项-v 允许在 BEGIN 语句中处理命令行变量。从命令行传递的每个变量前面都必须加一个-v 选项。

字段变量 字段变量可以像用户自定义的变量一样使用，唯一的区别是它们引用了字段。新的字段可以通过赋值来创建。字段变量引用的字段如果没有值，则被赋值为空串。字段的值发生变化时，awk 会以 OFS 的值作为字段分隔符重新计算$0 变量的值。字段数目通常被限制在 100 以内。

范例 6-88

```
% nawk ' { $5 = 1000 * $3 / $2; print } ' filename
```

说明

如果不存在第 5 个字段($5)，awk 将创建它并将表达式 1000 * $3 / $2 的结果赋给它。如果存在第 5 个字段，就直接将表达式的结果赋给它，覆盖该字段原来的内容。

范例 6-89

```
% nawk ' $4 == "CA" { $4  = "California"; print}'  filename
```

说明

如果第 4 个字段($4)匹配字符串 CA，awk 就将其重新赋值为 California。双引号是必需的，如果没有这对双引号，字符串 CA 就会被当成一个初始值为空的用户自定义变量。

内置变量 内置变量的名字都是大写的。它们可以被用于表达式，也可以被重置。请参见表 6-12 中所列的内置变量。

表 6-12 内置变量

变 量 名	含 义
ARGC	命令行参数的数目
ARGIND	命令行中当前文件在 ARGV 内的索引(仅用于 gawk)

(续表)

变 量 名	含 义
ARGV	命令行参数构成的数组
CONVFMT	数字转换格式，默认为%.6g(仅用于 gawk)
ENVIRON	包含当前 shell 环境变量值的数组
ERRNO	当使用 getline 函数进行读操作或者使用 close 函数时，因重定向操作而产生的系
FIELDWIDTHS	在分隔固定宽度的列表时，使用空白而不是 FS 进行分隔的字段宽度列表(仅用于
FILENAME	当前输入文件的文件名
FNR	当前文件的记录数
FS	输入字段分隔符，默认为空格
IGNORECASE	在正则表达式和字符串匹配中不区分大小写(仅用于 gawk)
NF	当前记录中的字段数
NR	目前的记录数
OFMT	数字的输出格式
OFS	输出字段分隔符
ORS	输出记录分隔符
RLENGTH	match 函数匹配到的字符串的长度
RS	输入记录分隔符
RSTART	match 函数匹配到的字符串的偏移量
RT	记录终结符，对于匹配字符或者用 RS 指定的 regex，gawk 将 RT 设置到输入文本
SUBSEP	数组下标分隔符

范例 6-90

```
(employees 数据库)
% cat employees2
    Tom Jones:4423:5/12/66:543354
    Mary Adams:5346:11/4/63:28765
    Sally Chang:1654:7/22/54:650000
    Mary Black:1683:9/23/44:336500

    (命令行)
    % nawk  -F:  '$1 == "Mary Adams"{print NR, $1, $2, $NF}' employees2

    (输出)
    2  Mary Adams 5346  28765
```

说明

-F 选项把字段分隔符设置为冒号。print 函数依次打印出记录号、第 1 个字段、第 2 个字段和最后一个字段($NF)。



范例 6-91

```
(employees 数据库)
% cat employees2
    Tom Jones:4423:5/12/66:543354
    Mary Adams:5346:11/4/63:28765
    Sally Chang:1654:7/22/54:650000
    Mary Black:1683:9/23/44:336500

(命令行)
% gawk -F: '{IGNORECASE=1}; \
  $1 == "mary adams"{print NR, $1, $2,$NF}' employees2

(输出)
2 Mary Adams 5346 28765
```

说明

-F 选项把字段分隔符设置为冒号。若 gawk 的内置变量 IGNORECASE 为非 0 值，则在正则表达式和字符串匹配中不区分大小写。接着，字符串 mary adams 匹配(写成 Mary Adams 也没关系)。最后，print 函数依次打印出记录号、第 1 个字段、第 2 个字段和最后一个字段($NF)。

6.13.3 BEGIN 模式

BEGIN 模式后面跟了一个操作块。awk 必须在对输入文件进行任何处理之前先执行该操作块。实际上，不需要任何输入文件，也能对 BEGIN 块进行测试，因为 awk 要在执行完 BEGIN 操作块后才开始读取输入。BEGIN 操作常常被用于修改内置变量(OFS、RS、FS 等)的值、为用户自定义变量赋初值和打印输出的页眉或标题。

范例 6-92

```
% nawk 'BEGIN{FS=":"; OFS="\t"; ORS="\n\n"}{print $1,$2,$3}' file
```

说明

在处理输入文件之前，nawk 先把字段分隔符(FS)设为冒号，把输出分隔符(OFS)设为制表符，还把输出记录分隔符(ORS)设为两个换行符。如果 BEGIN 的操作块中有两条或两条以上语句，必须用分号分隔它们或每行只写一条语句(在 shell 的命令提示符下输入时，必须用反斜杠来转义换行符)。

范例 6-93

```
% nawk 'BEGIN{print "MAKE YEAR"}'
MAKE YEAR
```

说明

awk 将显示 MAKE YEAR。awk 打开输入文件之前先执行该 print 函数，即使没有指定输入文件，awk 也照样打印 MAKE 和 YEAR。调试 awk 脚本时，可以先测试好 BEGIN 块的操作，再编写程序的其余部分。

6.13.4 END 模式

END 模式不匹配任何输入行，而是执行任何与之关联的操作。awk 处理完所有输入行之后才处理 END 模式。

范例 6-94

```
% nawk 'END{print "The number of records is " NR }' filename
The number of records is 4
```

说明

awk 处理完整个文件后才开始执行 END 块。此时 NR 的值是最后这条记录的记录号。

范例 6-95

```
% nawk '/Mary/{count++}END{print "Mary was found " count " times."}'
  employees
Mary was found 2 times.
```

说明

每遇到一个包含模式 Mary 的行，用户自定义的变量 counter 的值就加 1。awk 处理完整个文件后，END 块打印字符串 Mary was found，再跟上变量 count 的值和字符串 times。

6.14 重定向和管道

6.14.1 输出重定向

将 awk 的输出重定向到 UNIX/Linux 文件时，会用到 shell 的重定向操作符。重定向的目标文件名必须用双引号括起来。如果使用的重定向操作符为>，则文件被打开并清空。文件一旦打开，就会保持打开状态直至显示关闭或 awk 程序终止。此后 print 语句的输出都将追加到文件尾部。

符号>>也用于打开文件，但是不清除文件内容，它只向文件追加内容。

范例 6-96

```
% nawk '$4 >= 70 {print $1, $2  > "passing_file" }' filename
```

说明

如果记录的第 4 个字段的值大于或等于 70，它的头两个字段就被打印到文件 passing_file 中。

6.14.2 输入重定向(getline)

函数 getline　getline 函数用于从标准输入、管道或文件(非当前处理的文件)读取输入。



getline 函数用于读取下一输入行，并且设置内置变量 NF、NR 和 FNR。如果读到一条记录，函数就返回 1，如果读到 EOF(end of file，文件末尾)则返回 0。如果发生错误，比如打开文件失败，则 getline 函数返回 -1。

范例 6-97

```
% nawk 'BEGIN{ "date" | getline d; print d}' filename
Thu Jan 14 11:24:24 PST 2004
```

说明

先执行 UNIX/Linux 的 date 命令，将输出通过管道发给 getline，再通过 getline 将传来的内容赋值给用户自定义的变量 d，然后打印 d。

范例 6-98

```
% nawk 'BEGIN{ "date " | getline d; split( d, mon) ; print mon[2]}' filename
Jan
```

说明

先执行 date 命令，将输出通过管道发给 getline，接着，getline 从管道读取输入，然后保存在用户自定义变量 d 中。split 函数从 d 中生成一个名为 mon 的数组。最后，程序打印出数组 mon 的第 2 个元素。

范例 6-99

```
% nawk 'BEGIN{while("ls" | getline) print}'
a.out
db
dbook
getdir
file
sortedf
```

说明

ls 命令的输出将传递给 getline；每循环一次，getline 就从 ls 的输出中读取一行，并将其显示到屏幕上。不需要输入文件，因为 awk 会在文件打开之前先处理完 BEGIN 块。

范例 6-100

(命令行)

```
1   % nawk 'BEGIN{ printf "What is your name?" ;\
    getline name < "/dev/tty"}\
2   $1 ~ name {print "Found " name " on line ", NR "."}\
3   END{print "See ya, " name "."}' filename
```

(输出)

```
What is your name?  Ellie     < Waits for input from user >
Found Ellie on line 5.
See ya, Ellie.
```

说明

1. 在屏幕上显示 What is your name?，然后等待用户响应，getline 函数将从终端

(/dev/tty)接收输入，直到用户换行，然后，将输入保存在用户自定义的变量 name 中。

2. 如果第一个字段匹配之前赋给 name 的值，则执行 print 函数。

3. END 语句打印出“See ya,”，然后显示 Ellie(保存在变量 name 中的值)，再跟上一个句点。

范例 6-101

(命令行)
```
% nawk 'BEGIN{while (getline < "/etc/passwd"  > 0 )lc++; print lc}' file
```

(输出)
```
16
```

说明

awk 将逐行读取文件/etc/passwd，lc 随之递增直至到达 EOF，然后打印 lc 的值，即文件 passwd 的行数。

注意，如果文件不存在，getline 的值将是-1。如果读到文件尾，返回值是 0，而读到一行时，返回值则是 1。因此，命令

```
while (getline < "/etc/junk")
```

遇到文件/etc/junk 不存在的情况时，会进入死循环，因为返回值-1 导致条件为真。

6.15 管道

如果在 awk 程序中打开了管道，就必须先关闭它才能打开另一个管道。管道符右边的命令被括在双引号之间。每次只能打开一个管道。

范例 6-102

(数据库)
```
% cat names
john smith
alice cheba
george goldberg
susan goldberg
tony tram
barbara nguyen
elizabeth lone
dan savage
eliza goldberg
john goldenrod
```

(命令行)
```
% nawk '{print $1, $2 | "sort -r +1 -2 +0 -1 "}' names
```

(输出)
```
tony tram
john smith
```




```
dan savage
barbara nguyen
elizabeth lone
john goldenrod
susan goldberg
george goldberg
eliza goldberg
alice cheba
```

说明

awk 使用管道将 print 语句的输出结果发给 UNIX 的 sort 命令作为输入。sort 命令将以第 2 个字段为主键、第 1 个字段为次键对输入进行逆排序，例如，按姓氏进行逆排序。这种情况下，UNIX 命令必须被双引号括起来(参见附录 A 中的"sort")。

关闭文件和管道

如果打算再次在 awk 程序中使用某个文件或管道进行读写，则可能要先关闭程序，因为其中的管道会保持打开状态直至脚本运行结束。注意，管道一旦被打开，就会保持打开状态直至 awk 退出。因此，END 块中的语句也会受管道的影响。下面这个例子中，END 块的第一行命令将用来关闭管道。

范例 6-103

```
(脚本)
1   { print $1, $2, $3 | " sort -r +1 -2 +0 -1"}
    END{
2   close("sort -r +1 -2 +0 -1")
    <rest of statements>  }
```

说明

1. awk 把输入文件的每一行记录都通过管道发给 UNIX/Linux 的实用程序 sort。

2. 执行到 END 块时，管道被关闭。双引号中的字符串必须与最初打开管道的 pipe 命令字符串完全一致。

system 函数　awk 的内置函数 system 以 UNIX/Linux 的系统命令作为参数，执行该命令并且将命令的退出状态返回给 awk 程序。它很像 C 语言的一个标准库函数，该函数恰巧也为 system()。注意，作为参数的 UNIX/Linux 命令必须加双引号。

格式

```
system("UNIX Command")
```

范例 6-104

```
(脚本)
    {
1   system ( "cat" $1 )
2   system ( "clear" )
    }
```

说明

1. system 函数以 UNIX/Linux 的 cat 命令和输入文件的第 1 个字段作为参数。cat 命令

把第 1 个字段的值，即一个文件名，作为参数。UNIX/Linux shell 可以执行 cat 命令。

2. system 函数以 UNIX/Linux 的 clear 命令作为参数。shell 将执行 clear 命令，清空屏幕。

6.16 回顾

本节范例，除特别声明外，都使用了下面这个重复出现过多次的 datafile 文件。

```
% cat datafile
northwest   NW   Joel Craig      3.0   .98   3   4
western     WE   Sharon Kelly    5.3   .97   5   23
southwest   SW   Chris Foster    2.7   .8    2   18
southern    SO   May Chin        5.1   .95   4   15
southeast   SE   Derek Johnson   4.0   .7    4   17
eastern     EA   Susan Beal      4.4   .84   5   20
northeast   NE   TJ Nichols      5.1   .94   3   13
north       NO   Val Shultz      4.5   .89   5   9
central     CT   Sheri Watson    5.7   .94   5   13
```

6.16.1 递增和递减运算符

范例 6-105

```
% nawk '/^north/{count += 1; print count}' datafile
1
2
3
```

说明

如果记录以正则表达式 north 开头，则创建用户自定义的变量 count，然后加 1 并打印它的值。

范例 6-106

```
% nawk '/^north/{count++; print count}' datafile
1
2
3
```

说明

自动递增运算符将用户自定义变量 count 加 1。然后 count 的值被打印出来。

范例 6-107

```
% nawk '{x = $7--; print "x = "x ", $7 = "$7}' datafile
```




```
x = 3, $7 = 2
x = 5, $7 = 4
x = 2, $7 = 1
x = 4, $7 = 3
x = 4, $7 = 3
x = 5, $7 = 4
x = 3, $7 = 2
x = 5, $7 = 4
x = 5, $7 = 4
```

说明

第 7 个字段($7)的值被赋给用户自定义变量 x 后，自动递减运算符将第 7 个字段减 1。该命令会打印出 x 以及第 7 个字段的值。

6.16.2 内置变量

范例 6-108

```
% nawk '/^north/{print "The record number is " NR}' datafile
The record number is 1
The record number is 7
The record number is 8
```

说明

如果记录以正则表达式 north 开头，则打印字符串“The record number is ”和 NR(记录号)的值。

范例 6-109

```
% nawk '{print NR, $0}' datafile
1 northwest        NW      Joel Craig        3.0    .98   3    4
2 western          WE      Sharon Kelly      5.3    .97   5   23
3 southwest        SW      Chris Foster      2.7    .8    2   18
4 southern         SO      May Chin          5.1    .95   4   15
5 southeast        SE      Derek Johnson     4.0    .7    4   17
6 eastern          EA      Susan Beal        4.4    .84   5   20
7 northeast        NE      TJ Nichols        5.1    .94   3   13
8 north            NO      Val Shultz        4.5    .89   5    9
9 central          CT      Sheri Watson      5.7    .94   5   13
```

说明

打印 NR 的值(即当前记录的记录号)和$0 的值(即当前记录的全部内容)。

% cat datafile

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17

(续表)

eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

范例 6-110

```
% nawk 'NR==2,NR==5{print NR, $0}' datafile
2 western        WE    Sharon Kelly      5.3   .97   5   23
3 southwest      SW    Chris Foster      2.7   .8    2   18
4 southern       SO    May Chin          5.1   .95   4   15
5 southeast      SE    Derek Johnson     4.0   .7    4   17
```

说明

如果 NR 的值在 2 和 5 之间(即从第 2 条记录到第 5 条记录)，打印记录号(NR)和记录本身($0)。

范例 6-111

```
% nawk '/^north/{print NR, $1, $2, $NF, RS}' datafile
1 northwest NW 4

7 northeast NE 13

8 north NO 9
```

说明

如果记录以正则表达式 north 开头，则打印该记录的记录号(NR)，后跟第 1 和第 2 个字段、最后一个字段的值(注意 NF 前有一个美元符号)和 RS 的值(换行符)。由于 print 函数默认了一个换行，而 RS 又多加了一个换行，所以记录的间距会加倍。

范例 6-112 和范例 6-113 使用了 datafile2 数据库。

```
% cat datafile2
Joel Craig:northwest:NW:3.0:.98:3:4
Sharon Kelly:western:WE:5.3:.97:5:23
Chris Foster:southwest:SW:2.7:.8:2:18
May Chin:southern:SO:5.1:.95:4:15
Derek Johnson:southeast:SE:4.0:.7:4:17
Susan Beal:eastern:EA:4.4:.84:5:20
TJ Nichols:northeast:NE:5.1:.94:3:13
Val Shultz:north:NO:4.5:.89:5:9
Sheri Watson:central:CT:5.7:.94:5:13
```

范例 6-112

```
% nawk -F: 'NR == 5{print NF}' datafile2
7
```



说明

命令行中的字段分隔符被-F 选项设置为冒号。如果其记录号为(NR)为 5，则打印该记录的字段数。

范例 6-113

```
% nawk 'BEGIN{OFMT="%.2f";print 1.2456789,12E-2}' datafile2
1.25 0.12
```

说明

设置 print 函数的输出格式变量 OFMT 为：浮点数精确到小数点后两位。然后用新设置的格式打印 1.2456789 和 12E-2 这两个数。

```
% cat datafile
northwest    NW    Joel Craig       3.0    .98    3    4
western      WE    Sharon Kelly     5.3    .97    5    23
southwest    SW    Chris Foster     2.7    .8     2    18
southern     SO    May Chin         5.1    .95    4    15
southeast    SE    Derek Johnson    4.0    .7     4    17
eastern      EA    Susan Beal       4.4    .84    5    20
northeast    NE    TJ Nichols       5.1    .94    3    13
north        NO    Val Shultz       4.5    .89    5    9
central      CT    Sheri Watson     5.7    .94    5    13
```

范例 6-114

```
% nawk '{$9 = $6 * $7; print $9}' datafile
2.94
4.85
1.6
3.8
2.8
4.2
2.82
4.45
4.7
```

说明

第 6 个字段($6)和第 7 个字段($7)的乘积被保存到一个新的字段($9)里，然后打印出来。命令执行之前记录有 8 个字段，之后则有 9 个。

范例 6-115

```
% nawk '{$10 = 100; print NF, $9, $0}' datafile
10  northwest     NW     Joel Craig      3.0   .98   3   4    100
10  western       WE     Sharon Kelly    5.3   .97   5   23   100
10  southwest     SW     Chris Foster    2.7   .8    2   18   100
10  southern      SO     May Chin        5.1   .95   4   15   100
```

```
10  southeast   SE    Derek Johnson   4.0   .7    4   17    100
10  eastern     EA    Susan Beal      4.4   .84   5   20    100
10  northeast   NE    TJ Nichols      5.1   .94   3   13    100
10  north       NO    Val Shultz      4.5   .89   5   9     100
10  central     CT    Sheri Watson    5.7   .94   5   13    100
```

说明

每条记录的第 10 个字段都被赋值为 100，这是一个新字段。第 9 个字段不存在，因而被认为是空字段。输出结果是打印记录的字段数(NF)，后跟$9 的值(空字段)和整条记录($0)。第 10 个字段的值是 100。

6.16.3 BEGIN 模式

范例 6-116

```
% nawk 'BEGIN{print "---------EMPLOYEES---------"}'
---------EMPLOYEES---------
```

说明

BEGIN 模式带有一个操作块，其操作是在打开输入文件之前打印字符串"---------EMPLOYEES---------"。注意，本例的命令行并未提供输入文件，但 awk 并不受影响，因为 awk 首先执行的是 BEGIN 中的命令，而不是查找输入文件。

范例 6-117

```
% nawk 'BEGIN{print "\t\t---------EMPLOYEES-------\n"}\
  {print $0}' datafile
                    ---------EMPLOYEES-------
northwest        NW    Joel Craig        3.0     .98   3    4
western          WE    Sharon Kelly      5.3     .97   5    23
southwest        SW    Chris Foster      2.7     .8    2    18
southern         SO    May Chin          5.1     .95   4    15
southeast        SE    Derek Johnson     4.0     .7    4    17
eastern          EA    Susan Beal        4.4     .84   5    20
northeast        NE    TJ Nichols        5.1     .94   3    13
north            NO    Val Shultz        4.5     .89   5    9
central          CT    Sheri Watson      5.7     .94   5
```

说明

BEGIN 操作块最先被执行，于是打印出标题"---------EMPLOYEES---------"。第 2 个操作块打印输入文件中的每一条记录。当命令需换行时，可以用反斜杠来取消回车，且在分号或花括号处进行。

范例 6-118 使用了下面的 datafile2 数据库。

```
% cat datafile2
Joel Craig:northwest:NW:3.0:.98:3:4
Sharon Kelly:western:WE:5.3:.97:5:23
Chris Foster:southwest:SW:2.7:.8:2:18
```



(续表)

```
May Chin:southern:SO:5.1:.95:4:15
Derek Johnson:southeast:SE:4.0:.7:4:17
Susan Beal:eastern:EA:4.4:.84:5:20
TJ Nichols:northeast:NE:5.1:.94:3:13
Val Shultz:north:NO:4.5:.89:5:9
Sheri Watson:central:CT:5.7:.94:5:13
```

范例 6-118

```
% nawk 'BEGIN{ FS=":";OFS="\t"};/^Sharon/{print $1, $2, $8 }' datafile2
Sharon Kelly      western    28
```

说明

BEGIN 操作块被用来初始化变量。变量 FS(字段分隔符)被设为冒号，变量 OFS(输出字段分隔符)则被设为制表符(\t)。处理完 BEGIN 操作块中的内容后，awk 就打开 datafile2 文件并从中读取记录。如果某条记录以正则表达式 Sharon 开头，则打印它的第 1、2、8 字段($1，$2，$8)。输出结果的字段以制表符分隔。

6.16.4 END 模式

范例 6-119 和范例 6-120 使用的是 datafle 数据库。

```
% cat datafile
Northwest   NW   Joel Craig      3.0   .98   3   4
Western     WE   Sharon Kelly    5.3   .97   5   23
Southwest   SW   Chris Foster    2.7   .8    2   18
southern    SO   May Chin        5.1   .95   4   15
southeast   SE   Derek Johnson   4.0   .7    4   17
eastern     EA   Susan Beal      4.4   .84   5   20
northeast   NE   TJ Nichols      5.1   .94   3   13
north       NO   Val Shultz      4.5   .89   5   9
central     CT   Sheri Watson    5.7   .94   5   13
```

范例 6-119

```
% nawk 'END{print "The total number of records is " NR}' datafile
The total number of records is 9
```

说明

awk 处理完输入文件后，就开始执行 END 块中的语句：打印字符串“The total number of record is ”，后面跟上 NR 的值，即最后一条记录的记录号。

范例 6-120

```
% nawk '/^north/{count++}END{print count}' datafile
3
```

说明

如果记录以正则表达式 north 开头，用户自定义变量 count 就加 1。awk 处理完输入文件后，打印变量 count 的值。

6.16.5 包含 BEGIN 和 END 模式的 awk 脚本

范例 6-121 使用的是下面的 datafile2 文件。

```
% cat datafile2
Joel Craig:northwest:NW:3.0:.98:3:4
Sharon Kelly:western:WE:5.3:.97:5:23
Chris Foster:southwest:SW:2.7:.8:2:18
May Chin:southern:SO:5.1:.95:4:15
Derek Johnson:southeast:SE:4.0:.7:4:17
Susan Beal:eastern:EA:4.4:.84:5:20
TJ Nichols:northeast:NE:5.1:.94:3:13
Val Shultz:north:NO:4.5:.89:5:9
Sheri Watson:central:CT:5.7:.94:5:13
```

范例 6-121

```
   # Second awk script-- awk.sc2
1  BEGIN{ FS=":"
     print "  NAME\t\tDISTRICT\tQUANTITY"
    print "___________________________________________________\n"
   }

2    {print $1"\t  " $3"\t\t" $7}
    {total+=$7}
     /north/{count++}

3  END{
     print "---------------------------------------------"
     print "The total quantity is " total
     print "The number of northern salespersons is " count "."
   }
```

(输出)

```
4  % nawk -f awk.sc2 datafile2
    NAME DISTRICT  QUANTITY
   ___________________________________________
   Joel Craig      NW      4
```



```
Sharon Kelly      WE      23
Chris Foster      SW      18
May Chin          SO      15
Derek Johnson     SE      17
Susan Beal        EA      20
TJ Nichols        NE      13
Val Shultz        NO      9
Sheri Watson      CT      13
---------------------------------------------
The total quantity is 132
The number of northern salespersons is 3.
```

说明

1. BEGIN 块最先执行：设置字段分隔符(FS)，并打印输出的表头。

2. awk 脚本的正文部分包含的语句对来自输入文件 datafile2 的每一行都要执行一遍操作。

3. END 块中的语句是在输入文件关闭之后，即 awk 退出之前执行。

4. nawk 程序是在命令行上执行的。-f 选项后面跟着脚本的名字 awk.sc2，再往后则是输入文件的名字 datafile2。

本节其他的例子都使用的是下面的 datafile 文件。

```
% cat datafile
northwest   NW   Joel Craig      3.0   .98   3   4
western     WE   Sharon Kelly    5.3   .97   5   23
southwest   SW   Chris Foster    2.7   .8    2   18
southern    SO   May Chin        5.1   .95   4   15
southeast   SE   Derek Johnson   4.0   .7    4   17
eastern     EA   Susan Beal      4.4   .84   5   20
northeast   NE   TJ Nichols      5.1   .94   3   13
north       NO   Val Shultz      4.5   .89   5   9
central     CT   Sheri Watson    5.7   .94   5   13
```

6.16.6 printf 函数

范例 6-122

```
% nawk '{printf "$%6.2f\n",$6 * 100}' datafile
$ 98.00
$ 97.00
$ 80.00
$ 95.00
$ 70.00
$ 84.00
$ 94.00
$ 89.00
$ 94.00
```

说明

printf 函数将浮点数的格式设置为：右对齐(默认格式)，总长度为 6 位，其中小数点占一位，小数点右边占两位。第 6 个字段的值将在四舍五入后被打印。

范例 6-123

```
% nawk '{printf "|%-15s|\n",$4}' datafile
|Craig          |
|Kelly          |
|Foster         |
|Chin           |
|Johnson        |
|Beal           |
|Nichols        |
|Shultz         |
|Watson         |
```

说明

打印一个左对齐、长度为 15 的字符串。第 4 个字段被打印在两个竖杠之间，竖杠用来标明打印的宽度。

6.16.7 重定向与管道

范例 6-124

```
% nawk '/north/{print $1, $3, $4 > "districts"}' datafile
% cat districts
northwest Joel Craig
northeast TJ Nichols
north Val Shultz
```

说明

如果记录中包含正则表达式 north，则将其第 1、3、4 字段($1, $3, $4)打印到输出文件 districts 中。文件被打开后，就保持打开状态直至被关闭或程序终止。文件名“districts”必须加双引号。

范例 6-125

```
% nawk '/south/{print $1, $2, $3 >> "districts"}' datafile
% cat districts
southwest SW Chris
southern SO May
southeast SE Derek
```

说明

如果记录中包含模式 south，则将其第 1、2、3 字段($1，$2，$3)追加到输出文件 districts 的尾部。



% cat datafile

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17
eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

6.16.8 打开和关闭管道

范例 6-126

```
# awk script using pipes -- awk.sc3
1  BEGIN{
2      printf " %-22s%s\n", "NAME", "DISTRICT"
       print "--------------------------------------"

3  }
4  /west/{count++}
5  {printf "%s %s\t\t%-15s\n", $3, $4, $1| "sort +1" }
6  END{
7      close "sort +1"
       printf "The number of sales persons in the western "
       printf "region is " count "."    }
```

(输出)

```
   % nawk -f awk.sc3 datafile
1  NAME                    DISTRICT
2  --------------------------------------------------
3  Susan Beal              eastern
   May Chin                southern
   Joel Craig              northwest
   Chris Foster            southwest
   Derek Johnson           southeast
   Sharon Kelly            western
   TJ Nichols              northeast
   Val Shultz              north
   Sheri Watson            central
   The number of sales persons in the western region is 3.
```

说明

1. 专用模式 BEGIN 后跟的是一个操作块。该操作块中的语句最先被执行，且在 awk 处理输入文件之前。

2. printf 函数把字符串 NAME 显示为一个长度为 22、左对齐的字符串，跟在后面的字符串 DISTRICT 则是右对齐。

3. BEGIN 块结束。

4. 现在 awk 开始逐行处理输入文件。如果在记录中匹配模式 west，则执行这个操作块，即用户自定义的变量 count 加 1。awk 在第一次遇到变量 count 时将先创建它，并赋给它初值 0。

5. printf 函数用于将输出格式化并发送给管道。所有输出集齐后，被一同发送给 sort 命令。

6. END 块起始位置。

7. 必须用与打开时完全相同的命令来关闭管道(sort +1)，本例中所用的命令是“sort +1”。否则，END 块中的语句将与前面的输出一起被排序。

习题 5 nawk 练习

(参见从本书合作站点下载的文件中名为 lab5.data 的数据库文件)

Mike Harrington:(510) 548-1278:250:100:175
Christian Dobbins:(408) 538-2358:155:90:201
Susan Dalsass:(206) 654-6279:250:60:50
Archie McNichol:(206) 548-1348:250:100:175
Jody Savage:(206) 548-1278:15:188:150
Guy Quigley:(916) 343-6410:250:100:175
Dan Savage:(406) 298-7744:450:300:275
Nancy McNeil:(206) 548-1278:250:80:75
John Goldenrod:(916) 348-4278:250:100:175
Chet Main:(510) 548-5258:50:95:135
Tom Savage:(408) 926-3456:250:168:200
Elizabeth Stachelin:(916) 440-1763:175:75:300

上面这个数据库的记录内容包括姓名、电话号码和最近 3 个月的竞选捐款数额。

试编写一个能产生如下输出的 nawk 脚本。

```
% nawk -f nawk.sc db

                    ***CAMPAIGN 1998 CONTRIBUTIONS***
-----------------------------------------------------------------------
NAME                PHONE           Jan  |  Feb  |  Mar  |  Total Donated
-----------------------------------------------------------------------
Mike Harrington     (510) 548-1278  250.00  100.00  175.00  525.00
Christian Dobbins   (408) 538-2358  155.00  90.00   201.00  446.00
Susan Dalsass       (206) 654-6279  250.00  60.00   50.00   360.00
Archie McNichol     (206) 548-1348  250.00  100.00  175.00  525.00
Jody Savage         (206) 548-1278  15.00   188.00  150.00  353.00
Guy Quigley         (916) 343-6410  250.00  100.00  175.00  525.00
Dan Savage          (406) 298-7744  450.00  300.00  275.00  1025.00
Nancy McNeil        (206) 548-1278  250.00  80.00   75.00   405.00
```



```
John Goldenrod       (916) 348-4278  250.00  100.00  175.00  525.00
Chet Main            (510) 548-5258  50.00   95.00   135.00  280.00
Tom Savage           (408) 926-3456  250.00  68.00   200.00  618.00
Elizabeth Stachelin  (916) 440-1763  175.00  75.00   300.00  550.00
-------------------------------------------------------------------------
                            SUMMARY
-------------------------------------------------------------------------
The campaign received a total of $6137.00 for this quarter.
The average donation for the 12 contributors was $511.42.
The highest contribution was $300.00.
The lowest contribution was $15.00.
```

6.17 条件语句

awk 的条件语句源自 C 语言，可以用它们对包含判断语句的程序进行控制。

6.17.1 if 语句

以 if 结构开头的语句属于操作语句。条件模式(conditional pattern)中，if 是隐含的。而条件操作语句的 if 则是直接声明的，后面跟了一个用圆括号括起来的表达式。如果该表达式的运算结果为真(非 0 或非空)，则执行表达式后的语句(或语句块)。如果跟在条件表达式后面的语句不止一条，就要用分号或换行符把它们隔开，还要用花括号把这一组语句都括起来，以作为一个块来被执行。

格式

```
if (表达式) {
  语句; 语句; ……
}
```

范例 6-127

```
1  % nawk '{if ( $6 > 50 ) print $1 "Too high"}' filename
2  % nawk '{if ($6 > 20 && $6  <= 50){safe++; print "OK"}}' filename
```

说明

1. 在 if 操作块中对表达式进行测试。如果第 6 个字段的值大于 50，就执行打印语句。由于跟在表达式后面的是单条语句，所以不需要加花括号(filename 代表输入文件)。

2. 在 if 操作块中测试表达式。如果第 6 个字段的值大于 20 并且小于 50，就要将表达式后面的那些语句作为一个块来执行，因此，必须用花括号把它们括起来。

6.17.2 if/else 语句

if/else 语句实现双路判断。如果关键字 if 后面的表达式为真，就执行与该表达式关联的语句块。如果这个表达式的运算结果为假或 0，则执行关键字 else 后面的语句块。如果 if 或 else 包含多条语句，就必须用花括号把它们合成一个语句块。

格式

```
{if (表达式){
 语句; 语句; ……
 }
else {
 语句; 语句; ……
 }
}
```

范例 6-128

```
1  % nawk '{if( $6 > 50) print  $1 " Too high" ;\
     else print "Range is OK"}' filename
2  % nawk '{if ( $6 > 50 ) { count++; print $3 } \
     else { x+5; print $2 } }' filename
```

说明

1. 如果第一个表达式为真，即第 6 个字段($6)的值大于 50，则 print 函数打印第 1 个字段和字符串“Too high”。否则就执行 else 后的语句，打印字符串“Range is OK”。

2. 如果第一个表达式为真，即第 6 个字段($6)的值大于 50，则执行表达式后面的这个语句块。否则就执行 else 后面的那个语句块。注意，语句块必须括在花括号中。

6.17.3 if/else 和 else if 语句

if/else 和 else if 语句提供了多路判断功能。如果跟在关键字 if 后的表达式为真，则执行与该表达式关联的语句块，同时，程序的控制流将跳到与最后一个 else 关联的最后一个右花括号后，从这个位置继续往下执行。否则，控制转到 else if，测试与其关联的表达式。如果第一个 else if 的条件为真，则执行对应表达式后的语句。如果 else if 的条件表达式都不为真，控制就转到 else 语句。这个 else 被称作默认操作，因为只要其他语句都不为真，就执行该 else 块。

格式

```
{if (表达式){
 语句; 语句; ……
 }
else if (表达式){
 语句; 语句; ……
 }
else if (表达式){
 语句; 语句; ……
 }
else {
 语句; 语句; ……
 }
}
```



范例 6-129

(脚本)

```
1   {if ( $3 > 89 && $3 < 101 ) Agrade++
2   else if ( $3 > 79 ) Bgrade++
3   else if ( $3 > 69 ) Cgrade++
4   else if ( $3 > 59 ) Dgrade++
5   else Fgrade++
    }
    END{print "The number of failures is" Fgrade }
```

说明

1. if 语句是一个操作，因此必须用花括号括起来。表达式的计算是从左向右进行。如果第一个表达式为假，则整个表达式为假。如果第一个表达式为真，则计算符号逻辑与(&&)后面的那个表达式。如果整个表达式为真，则变量 Agrade 加 1。

2. 如果关键字 if 后面的表达式值为假，就测试这个 else if 的表达式。如果该表达式的值为真，就执行它后面的语句。也就是说，如果第 3 个字段($3)的值大于 79，则变量 Bgrade 加 1。

3. 如果头两个条件语句都为假，就测试这个 else if 表达式，如果第 3 个字段($3)的值大于 69，则将变量 Cgrade 加 1。

4. 如果头三个条件语句都为假，就测试这个 else if 表达式，如果第 3 个字段($3)的值大于 59，则将变量 Dgrade 加 1。

5. 如果上面的表达式都不为真，就执行 else 块，将变量 Fgrade 加 1。接下来的花括号将结束整个操作块。

6.18 循环

循环的功能是：当测试表达式的条件为真时，重复执行表达式后面的语句。循环常常被用来对记录中的每个字段重复执行某种操作，或者在 END 块中用来循环处理某个数组中的所有元素。awk 有 3 种类型的循环：while 循环、for 循环和特殊 for 循环，特殊 for 循环将在稍后介绍 awk 数组时讨论。

6.18.1 while 循环

使用 while 循环的第一步是给一个变量设初值，然后在 while 表达式中测试该变量。如果求得表达式的值为真(非 0)，则进入循环体执行其中的语句。如果循环体内有多条语句，就必须用花括号把这些语句括起来。循环块结束之前，一定要更新用来控制循环表达式的变量，否则循环将无休止地进行下去。下面这个例子中，每处理一条新记录，循环控制变量就会被重置一次。

do/while 循环与 while 循环很相似，唯一的区别在于 do/while 要先执行循环体至少一次，然后才测试表达式。

范例 6-130

```
% nawk '{ i  = 1; while ( i <= NF ) { print NF, $i ; i++ } }' filename
```

说明

变量 i 被初始化为 1；当 i 小于或等于记录的字段数(NF)时，先执行 print 语句，然后将 i 加 1。接下来又重新测试表达式，直至 i 大于 NF 的值。变量 i 要在 awk 开始处理下一条记录时被重置。

6.18.2 for 循环

for 循环和 while 循环基本相同，只不过 for 循环的圆括号中需要 3 个表达式，前两个分别是初始化表达式和测试表达式，第 3 个则用于更新测试表达式所用的变量。在 awk 的 for 循环中，圆括号里的第一条语句只能初始化一个变量(C 语言中与之对应的语句则可以用逗号分隔的形式初始化多个变量)。

范例 6-131

```
% nawk '{ for( i = 1; i <= NF; i++) print NF,$i }' filex
```

说明

变量 i 被初始化为 1，然后测试它是否小于或等于记录的字段数目(NF)。若是，print 函数便打印出 NF 和$i 的值($i 代表第 i 个字段)，然后将 i 加 1(for 循环经常会在 END 操作中与数组一同使用，循环处理数组的所有元素。请参见 6.20 节，“数组”)。

6.18.3 循环控制

break 和 continue 语句 可以在某个特定条件为真时，使用 break 语句跳出循环。continue 语句的作用则是在特定条件为真时，让循环跳过 continue 之后语句，将控制转回循环顶部，开始下一轮循环。

范例 6-132

```
(脚本
1   {for ( x = 3; x <= NF; x++ )
        if ( $x < 0 ){ print "Bottomed out!"; break}
        # breaks out of for loop
    }

2   {for ( x = 3; x <= NF; x++ )
       if ( $x == 0 ) { print "Get next item"; continue}
       # starts next iteration of the for loop
    }
```

说明

1. 如果字段$x 的值小于 0，则 break 语句将控制跳转到循环体的右花括号后面的那条语句，即跳出循环。



2. 如果字段$x 的值等于 0，则 continue 语句使控制转回循环顶部并开始执行，将从 for 循环的第 3 个表达式 x++开始。

6.19 程序控制语句

6.19.1 next 语句

next 语句从输入文件中取出下一行输入，然后从 awk 脚本的顶部重新开始执行。

范例 6-133

```
(脚本)
{ if ($1 ~ /Peter/){next}
   else {print}
}
```

说明

如果某一行的第一个字段包含 Peter，awk 就跳过该行，从输入文件中读取下一行，然后从头开始执行脚本。

6.19.2 exit 语句

exit 语句用于终止 awk 程序。它只能中断对记录的处理，不能跳过 END 语句。如果 exit 语句的参数是一个 0~255 之间的值(exit 1)，这个值就会被打印在命令行上，以表明程序是否执行成功，并且指出失败的类型。

范例 6-134

```
(脚本)
  {exit (1) }

(命令行)
% echo $status     (csh)
1
$ echo $?    (sh/ksh)
1
```

说明

退出状态为 0 表示成功，退出状态非 0 则表示失败(这是 UNIX 的统一约定)。退出状态由程序员决定是否在程序中提供。在这个例子中，命令返回的退出状态值为 1。

6.20 数组

数组在 awk 中被称为关联数组(associative arrays)，因为它的下标既可以是数字也可以

是字符串。下标通常又被称作键(key)，并且与对应的数组元素的值相关联。数组元素的键和值都存储在 awk 程序内部的一个表中，该表采用的是散列算法。正是由于使用了散列算法，所以数组元素不是顺序存储的，如果将数组的内容显示出来，元素的排列顺序也许跟想象中的不一样。

和变量一样，数组也是被用到时才被创建，而且，awk 还能判定这个数组用于保存数字还是字符串。根据使用时的上下文环境，数组元素被初始化为数字 0 或空字符串。数组的大小不需要声明。awk 数组可用于从记录中收集信息，也可用于统计总数、计算词数、记录模式出现次数等应用。

6.20.1 关联数组的下标

使用变量作为数组索引请参见范例 6-135。

范例 6-135

```
(输出文件)
% cat employees
Tom Jones              4424     5/12/66              543354
Mary Adams             5346     11/4/63              28765
Sally Chang            1654     7/22/54              650000
Billy Black            1683     9/23/44              336500

(命令行)
1     % nawk '{name[x++]=$2};END{for(i=0; i<NR; i++)\
       print i, name[i]}' employees
      0 Jones
      1 Adams
      2 Chang
      3 Black

2     % nawk '{id[NR]=$3};END{for(x = 1; x <= NR; x++)\
       print id[x]}' employees
      4424
      5346
      1654
      1683
```

说明

1. 数组 name 的下标是用户自定义的变量 x。运算符++表明这是一个数值型的变量。awk 将 x 初始化为 0，并且每次使用 x 后将其加 1(所用的是后递增运算符)。每条记录的第 2 个字段都将赋值给数组 name 中的相应元素。END 块使用 for 循环来循环处理数组，将从下标 0 开始，依次打印数组元素的值。下标只是一个键，所以不必从 0 开始。下标可以从任意值开始，数字或字符串都可以。

2. awk 变量 NR 保存当前记录的记录号。本例用 NR 作为下标，把每条记录的第 3 个字段赋值给数组中的相应元素。最后，for 循环对数组进行循环处理，打印出保存在数组中的值。



特殊 for 循环 当下标为字符串或非连续的数字时，不能用 for 循环来遍历数组。这时候就要使用特殊 for 循环。特殊 for 循环把下标作为键来查找与之关联的值。

格式

```
(for (item in arrayname) {
  print arrayname[item]
  }
)
```

范例 6-136

(输入文件)

```
% cat db
1    Tom Jones
2    Mary Adams
3    Sally Chang
4    Billy Black
5    Tom Savage
6    Tom Chung
7    Reggie Steel
8    Tommy Tucker
```

(循环命令行)

```
1   % nawk '/^Tom/{name[NR]=$1};\
      END{for( i = 1; i <= NR; i++ )print name[i]}' db
    Tom



    Tom
    Tom

    Tommy
```

(特殊循环命令行)

```
2   % nawk '/^Tom/{name[NR]=$1};\
      END{for(i in name){print name[i]}}' db
    Tom
    Tommy
    Tom
    Tom
```

说明

1. 如果在输入行的行首匹配到正则表达式 Tom，就为数组 name 赋一个值。NR 值(当前记录号)，将作为 name 数组的索引。在每一行上匹配到 Tom 时，name 数组就赋一个第一个字段($1)的值。当到达 END 块时，name 数组仅包含 name[1]、name[5]、name[6]、name[8] 这 4 个元素。因此，当使用 for 循环打印 name 数组的值时，索引 2、3、4、7 为空。

2. 用特殊 for 循环遍历数组，只打印有相应下标的元素的值。打印结果的次序是随机的，因为关联数组是以散列方式存储的。

用字符串作为数组下标 数组下标可以由包含单个字符或字符串的变量组成，如果是字符串，则必须用双引号引起来。

范例 6-137

(输入文件)

```
% cat datafile3
tom
mary
sean
tom
mary
mary
bob
mary
alex
```

(脚本)

```
    # awk.sc script
1   /tom/ { count["tom"]++ }
2   /mary/ { count["mary"]++ }
3   END{print "There are " count["tom"] " Toms in the file and
      "count["mary"]" Marys in the file."}
```

(命令行)

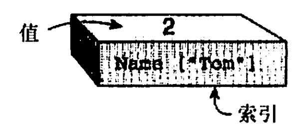
```
% nawk -f awk.sc datafile3
There are 2 Toms in the file and 4 Marys in the file.
```

说明

1. 数组 count 包含两个元素：count["tom"]和 count["mary"]。这两个数组元素的初值都是 0。每次匹配到 tom 时，数组元素 count["tom"]的值都加 1。

2. 同样的过程被应用于 count["mary"]。注意，每行只会算一次，即便 tom(或 mary)在该行中出现多次。

3. END 模式打印出每个数组元素的值。



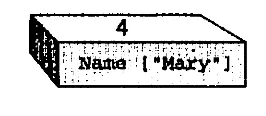


图 6-1 用字符串做数组下标(范例 6-137)

使用字段的值作为数组下标 任何表达式都可以用作数组的下标。所以，也可以用字段作下标。范例 6-138 中的程序用于计算所有名字在第 2 个字段出现的次数，并引入了一种 for 循环的新形式。

```
for( index_value in array ) statement
```

在前面介绍的例子中，END 块中出现的 for 循环的工作过程如下：变量 name 被设为



count 数组的索引值，在每次 for 循环的迭代中，执行 print 操作，首先打印的是索引值，然后是保存在元素中的值(打印输出的次序无法确定)。

范例 6-138

(输入文件)

```
% cat datafile4
4234  Tom     43
4567  Arch    45
2008  Eliza   65
4571  Tom     22
3298  Eliza   21
4622  Tom     53
2345  Mary    24
```

(命令行)

```
% nawk '{count[$2]++}END{for(name in count)print name,count[name] }'
  datafile4
Tom 3
Arch 1
Eliza 2
Mary 1
```

说明

这条 awk 语句首先用记录的第 2 个字段作为数组 count 的下标。数组的下标随第 2 个字段的变化而变化，所以数组 count 的第一个下标是 Tom，而 count["Tom"]中保存的值是 1。

然后，count["Arch"]、count["Eliza"]和 count["Mary"]相继被设为 1。当在第 2 个字段中再次出现 Tom 时，count["Tom"]的值将被加 1，于是它目前的值是 2。Arch、Eliza 和 Mary 再次出现时其过程类似。

范例 6-139

(输入文件)

```
% cat datafile4
4234  Tom     43
4567  Arch    45
2008  Eliza   65
4571  Tom     22
3298  Eliza   21
4622  Tom     53
2345  Mary    24
```

(命令行)

```
% nawk  '{dup[$2]++; if  (dup[$2]  > 1){name[$2]++ }}\
  END{print "The duplicates were"\
  for (i in name){print i, name[i]}}' datafile4
```

(输出)

```
Tom 2
Eliza 2
```

说明

数组 dup 的下标是第 2 个字段的值，即人名。dup 数组中元素的值最初都是 0，每处理一条记录，相应元素的值就加 1。如果名字重复出现，则对应该下标的元素值就会变成 2，并相应地逐渐增加。如果 dup 数组中某个元素的值大于 1，就会创建一个名为 name 的新数组，也是以第 2 个字段的值作为下标，用于记录出现次数大于 1 的人名。

数组与 split 函数　awk 的内置函数 split 能够将字符串拆分为词，然后保存在数组中。您可以指定字段分隔符，也可以就用 FS 的当前值。

格式

```
split(字符串, 数组, 字段分隔符)
split(字符串, 数组)
```

范例 6-140

(命令行)

```
% nawk BEGIN{ split( "3/15/2004", date, "/");\
  print "The month is " date[1] "and the year is "date[3]"} filename
```

(输出)

```
The month is 3 and the year is 2004.
```

说明

将字符串 3/15/2004 保存到数组 date 中，用正斜杠作为字段分隔符。现在 date[1]中是 3，date[2]中是 15，而 date[3]中则是 2004。字段分隔符用第 3 个参数指定，如未指定，就以 FS 的值做字段分隔符。

delete 函数　delete 函数用于删除数组元素。

范例 6-141

```
% nawk '{line[x++]=$2}END{for(x in line) delete(line[x])}' filename
```

说明

赋给数组 line 的值是第 2 个字段的值。所有记录都处理完后，特殊 for 循环将遍历数组的所有元素，并由 delete 函数来删除它们。

多维数组(nawk)　awk 虽然没有宣称支持多维数组，却提供了定义多维数组的方法。awk 定义多维数组的方法是把多个下标串成字符串，下标之间用内置变量 SUBSEP 的值分隔。变量 SUBSEP 的值是“\034”，这是个不可打印的字符，极少被使用，因此不太可能被用作下标中的字符。表达式 matrix[2, 8]其实就是数组 matrix[2 SUBSEP 8]，转换后所得的结果为 matrix["2\0348"]。因此，下标成了关联数组中的唯一标识符。

范例 6-142

(输入文件)

```
1 2 3 4 5
2 3 4 5 6
6 7 8 9 10
```



(脚本)
```
1    {nf=NF
2    for(x = 1; x <= NF; x++ ){
3       matrix[NR, x] = $x
        }
     }
4    END { for (x=1; x <= NR; x++ ){
        for (y = 1; y <= nf; y++ )
            printf "%d ", matrix[x,y]
     printf"\n"
        }
     }
```

(输出)
```
1 2 3 4 5
2 3 4 5 6
6 7 8 9 10
```

说明

1. 将 NF 的值(字段数)赋给变量 nf(该程序假定每条记录都是由 5 个字段组成)。

2. 进入 for 循环，依次把输入行每个字段的字段号保存到变量 x 中。

3. matrix 是一个二维数组。每个字段的值将赋给下标为 NR(当前记录的记录号)和 x 的数组元素。

4. END 块中的两个 for 循环被用来遍历 matrix 数组，并打印数组中保存的值。这个例子只是用来说明如何使用多维数组。

6.20.2 处理命令行参数(nawk)

ARGV　nawk(新版的 awk)可以从内置数组 ARGV 中得到命令行参数，其中包括命令 nawk。但所有传递给 nawk 的选项都不在其中。ARGV 数组的下标从 0 开始(以上内容只适用于 nawk)。

ARGC　ARGC 是一个包含命令行参数个数的内置变量。

范例 6-143

(脚本)
```
# Scriptname: argvs
BEGIN{
   for ( i=0; i < ARGC; i++ ){
      printf("argv[%d] is %s\n", i, ARGV[i])
      }
   printf("The number of arguments, ARGC=%d\n", ARGC)
}
```

(输出)
```
% nawk -f argvs datafile
argv[0] is nawk
argv[1] is datafile
The number of arguments, ARGC=2
```

说明

for 循环先将 i 设为 0，然后测试它是否小于命令行参数的个数(ARGC)，再用 printf 函数依次显示出每个参数。所有参数处理完之后，最后那条 printf 语句用来输出参数的个数 ARGC。这个例子说明 awk 并不把命令行选项视为参数。

范例 6-144

(命令行)

```
% nawk -f argvs datafile "Peter Pan" 12
argv[0] is nawk
argv[1] is datafile
argv[2] is Peter Pan
argv[3] is 12
The number of arguments, ARGC=4
```

说明

和上个例子一样，打印出所有参数。nawk 命令被当成第一个参数，而-f 选项和脚本文件名(argvs)则被排除在外。

范例 6-145

(数据库)

```
% cat datafile5
Tom Jones:123:03/14/56
Peter Pan:456:06/22/58
Joe Blow:145:12/12/78
Santa Ana:234:02/03/66
Ariel Jones:987:11/12/66
```

(脚本)

```
% cat arging.sc
# Scriptname: arging.sc
1   BEGIN{FS=":"; name=ARGV[2]
2   print "ARGV[2] is "ARGV[2]
    }
    $1 ~ name { print $0 }
```

(命令行)

```
% nawk -f arging.sc datafile5 "Peter Pan"
ARGV[2] is Peter Pan
Peter Pan:456:06/22/58
nawk: can't open Peter Pan
input record number 5, file Peter Pan
source line number 2
```

说明

1. 在 BEGIN 块中，ARGV[2]的值，即 Peter Pan，被赋给变量 name。

2. Peter Pan 被打印出来了，但是，处理完 datafile 并将其关闭后，awk 试图把 Peter Pan 作为输入文件打开。awk 把参数都作为输入文件。



范例 6-146

(脚本)

```
% cat arging2.sc
BEGIN{FS=":"; name=ARGV[2]
   print "ARGV[2] is " ARGV[2]
   delete ARGV[2]
}
$1  ~ name { print $0 }
```

(命令行)

```
% nawk -f arging2.sc datafile "Peter Pan"
ARGV[2] is Peter Pan
Peter Pan:456:06/22/58÷
```

说明

nawk 把 ARGV 数组的元素作为输入文件。且 nawk 用完一个参数就将它左移，接着处理下一个，直到 ARGV 数组变空。如果某个参数使用后立刻被删除，那么这个参数就不会被当作下一个输入文件来处理。

6.21 awk 的内置函数

字符串函数

sub 和 gsub 函数 sub 函数用于在记录中查找能够匹配正则表达式的最长且最靠左的子串，然后用替换串取代找到的子串。如果指定了目标串，就在目标串中查找能够匹配正则表达式的最长且最靠左的子串，并将找到的子串替换为替换串。若未指定目标串，则在整个记录中查找。

格式

```
sub (正则表达式, 替换串);
sub (正则表达式, 替换串, 目标串);
```

范例 6-147

```
1  % nawk '{sub(/Mac/, "MacIntosh"); print}' filename
2  % nawk '{sub(/Mac/, "MacIntosh", $1); print}' filename
```

说明

1. 在记录($0)中第一次匹配到正则表达式 Mac 时，Mac 被替换为字符串 MacIntosh。sub 函数只对每行中出现的第一个匹配字符串进行替换(请参见用于替换多次匹配的 gsub 函数)。

2. 在记录的第 1 个字段($1)中第一次匹配到正则表达式 Mac 时，Mac 被替换为字符串 MacIntosh。sub 函数只对目标串中出现的第一个匹配字符串进行替换。gsub 函数则对字符串中的正则表达式进行全局替换，即替换所有在记录($0)中出现的正则表达式。

格式

```
gsub (正则表达式, 替换串);
gsub (正则表达式, 替换串, 目标串);
```

范例 6-148

```
1 % nawk '{ gsub(/CA/, "California"); print }' datafile
2 % nawk '{ gsub(/[Tt]om/, "Thomas", $1 ); print }' filename
```

说明

1. 记录($0)中找到的每个正则表达式 CA 都被替换为 California。
2. 在第一个字段中找到的每个正则表达式 Tom 或 tom 都被替换为 Thomas。

index 函数 index 函数返回子串在字符串中第一次出现的位置。偏移量从位置 1 开始计算。

格式

```
index(字符串, 子串)
```

范例 6-149

```
% nawk '{ print index("hollow", "low") }' filename
4
```

说明

返回的数字是子串 low 在字符串 hollow 中第一次出现的位置，偏移量从 1 开始计算。

length 函数 length 函数返回字符串中字符的个数。如果未指定参数，则 length 函数返回记录中的字符个数。

格式

```
length(字符串)
length
```

范例 6-150

```
% nawk '{ print length("hello") }' filename
5
```

说明

length 函数返回字符串 hello 的字符个数。

substr 函数 substr 函数返回从字符串指定位置开始的一个子串。如果指定了子串的长度，则返回字符串的相应部分。如果指定的长度超出了字符串的实际范围，则返回其实际内容。

格式

```
substr (字符串, 起始位置)
substr (字符串, 起始位置, 子串长度)
```

范例 6-151

```
% nawk ' { print substr("Santa Claus", 7, 6 )} ' filename
```




```
Claus
```

说明

在字符串“Santa Claus”中，打印从位置 7 开始、长度为 6 个字符的子串。

match 函数　match 函数返回正则表达式在字符串中出现的位置，如果未出现，则返回 0。match 函数把内值变量 RSTART 设为子串在字符串中的起始位置，RLENGTH 则设为子串的长度。这些变量可以被 substr 函数用来提取相应模式的子串(只可用于 nawk)。

格式

```
match(字符串, 正则表达式)
```

范例 6-152

```
% nawk 'END{start=match("Good ole USA", /[A-Z]+$/); print start}'\
filename
10
```

说明

正则表达式/[A-Z]+$/的意思是查找在字符串尾部连续出现的大写字母。找到的子串 USA 是从字符串“Good ole USA”的第 10 个字符开始的。如果字符串未能匹配到正则表达式，则返回 0。

范例 6-153

```
1  % nawk 'END{start=match("Good ole USA", /[A-Z]+$/);\
    print RSTART, RLENGTH}' filename
   10 3

2  % nawk 'BEGIN{ line="Good ole USA"}; \
    END{ match( line, /[A-Z]+$/);\
    print substr(line, RSTART,RLENGTH)}' filename
   USA
```

说明

1. 变量 RSTART 被 match 函数设置为匹配到的正则表达式在字符串中的起始位置。变量 RLENGTH 则被设为子串的长度。

2. substr 函数在变量 line 中查找子串，把 RSTART 和 RLENGTH 的值(由 match 函数设置)作为子串的起始位置和长度。

split 函数　split 函数使用由第 3 个参数指定的字段分隔符，把字符串拆分成一个数组。如果没有提供第 3 个参数，awk 将把 FS 的当前值作为字段分隔符。

格式

```
split (字符串, 数组, 字段分隔符)
split (字符串, 数组)
```

范例 6-154

```
% awk 'BEGIN{split("12/25/2001",date,"/");print date[2]}' filename
25
```

说明

split 函数把字符串 12/25/2001 拆分为数组 date，以正斜杠作为字段分隔符。数组 date 的下标从 1 开始。awk 将打印数组 date 的第 2 个元素。

sprintf 函数 sprintf 函数返回一个指定格式的表达式。可以在 sprintf 函数中使用 printf 函数的格式规范。

格式

```
variable = sprintf("含有格式说明的字符串", 表达式1, 表达式2, ..., 表达式n)
```

范例 6-155

```
% awk '{line = sprintf ( "%-15s %6.2f ", $1 , $3 ); print line}' filename
```

说明

按照 printf 的规范设置第 1 个和第 3 个字段的格式(一个左对齐、长度为 15 的字符串和一个右对齐、长度为 6 个字符的浮点数)。结果被赋给用户自定义的变量 line。请参见 6.4.3 节，“printf 函数”。

6.22 内置算术函数

表 6-13 列出了 awk 的内置算术函数，表中的 x 和 y 是任意表达式。

表 6-13 算术函数

函数名	返回值
atan2(x, y)	值域内 y/x 的反正切
cos(x)	x 的余弦，x 为弧度
exp(x)	x 的 e 指数函数
int(x)	x 的整数部分，当 x>0，向下取整
log(x)	x 的自然对数(底数为 e)
rand()	随机数 r(0<r<1)
sin(x)	x 的正弦，x 为弧度
sqrt(x)	x 的平方根
srand(x)	x 是 rand()的新种子

6.22.1 整数函数

int 函数将舍去小数点后的所有数字，生成一个整数。int 函数不执行舍入操作。

范例 6-156

```
1   % awk 'END{print 31/3}' filename
```




```
    10.3333
2   % awk 'END{print int(31/3)}' filename
    10
```

说明

1. END 块将除法运算的结果打印成浮点数形式。

2. END 块中的 int 函数把除法运算的结果从小数点开始舍去，显示的结果是一个整数。

6.22.2 随机数发生器

rand 函数 rand 函数生成一个大于或等于 0、小于 1 的伪随机浮点数。

范例 6-157

```
% nawk '{print rand()}' filename
0.513871
0.175726
0.308634

% nawk '{print rand()}' filename
0.513871
0.175726
0.308634
```

说明

每次运行程序都打印出同一组数字。可以用 srand 函数可以为 rand 函数的种子设一个新值，否则，如上例所示，每次调用 rand 都只会重复出现同一数列。

srand 函数 如果未指定参数，srand 函数会根据当前时刻为 rand 函数生成一个种子。srand(x)则把 x 设成种子。通常，程序应该在运行过程中不断地改变 x 的值。

范例 6-158

```
% nawk 'BEGIN{srand()};{print rand()}' filename
0.508744
0.639485
0.657277

% nawk 'BEGIN{srand()};{print rand()}' filename
0.133518
0.324747
0.691794
```

说明

srand 函数为 rand 设置了一个新种子，起点是当前时刻。因此，每次调用 rand 都打印出一组新的数列。

范例 6-159

```
% nawk 'BEGIN{srand()};{print 1 + int(rand() * 25)}' filename
6
```

```
24
14
```

说明

srand 含数为 rand 设置了一个新种子，起点是当前时刻。rand 函数在 0~25 之间选出一个随机数，然后将其化为整数。

6.23 用户自定义函数(nawk)

脚本中凡是可以出现模式操作规则的位置都可以放置用户自定义的函数。

格式

```
函数名 ( 参数, 参数, 参数, … ) {
    语句
    return 表达式
  (注: return 语句和表达式都是可选项)
}
```

变量以参数值的方式传递，且仅在使用它的函数中局部有效。函数使用的只是变量的副本。数组则通过地址或引用被传递，因此，可以在函数中直接修改数组的元素。函数中的任何变量，只要不是从参数列表中传来的，就都被视为全局变量，也就是说，该变量对整个 awk 程序都是可见的，而且，如果它在函数中发生了改变，即在整个程序中发生了改变。在函数中提供局部变量的唯一途径就是将它加入参数列表中。这类参数通常放在参数列表的末端。当调用函数时，如果没有指定某个形参的值，该参数就会被初始化为空。return 语句会把控制权交还给调用者，可能还会返回一个值。

范例 6-160

```
(排序前命令行显示的 grades 文件)
% cat  grades
44 55 66 22 77 99
100 22 77 99 33 66
55 66 100 99 88 45

(脚本)
% cat sorter.sc
    # Scriptname: sorter
    # It sorts numbers in ascending order
1   function sort ( scores, num_elements, temp, i, j ) {
        # temp, i, and j will be local and private,
        # with an initial value of null.
2       for( i = 2; i <= num_elements ; ++i ) {
3           for ( j = i; scores [j-1] > scores[j]; --j ){
              temp = scores[j]
               scores[j] = scores[j-1]
               scores[j-1] = temp
            }
```

```
4        }
5   }
6   {for ( i = 1; i <= NF; i++)
        grades[i]=$i
7   sort(grades, NF)    # Two arguments are passed
8   for( j = 1; j <= NF; ++j )
        printf( "%d ", grades[j] )
        printf("\n")
    }
```

(排序后)

```
% nawk -f sorter.sc grades
22 44 55 66 77 99
22 33 66 77 99 100
45 55 66 88 99 100
```

说明

1. 定义名为 sort 的函数。函数定义可以出现在脚本的任意位置。除了那些作为参数传传递的变量外，所有其他变量的域都是全局的。即如果在函数中发生了变化，也就在整个 nawk 脚本中发生了变化。数组是通过引用进行传递的。圆括号中共有 5 个形参，其中：数组 scores 将通过引用被传递，所以，如果在函数中修改了这个数组中任何一个元素，原来的数组也会被修改。变量 num_elements 是一个局部变量，是原变量的一个副本。变量 temp，i 和 j 则是函数的局部变量。

2. 外层的 for 循环将遍历一个整数数组，前提是该数组中至少有两个整数可用于比较。

3. 内层的 for 循环用当前这个整数与数组中前一个整数(scores[j-1])进行比较。如果前一个整数大于当前这个整数，就把当前这个数组元素的值赋给变量 temp，然后把前一个元素的值赋给当前元素。

4. 外层循环至此结束。

5. 函数定义的末尾。

6. 脚本的第一个操作块由此开始。for 循环遍历当前记录的所有字段，生成一个整数数组。

7. 调用 sort 函数，把由当前记录生成的整数数组和当前记录的字段数作为参数传给它。

8. sort 函数结束后，程序控制由此开始。这个 for 循环用于打印完成排序的数组中的元素。

6.24 复习

如果没有特别说明，本节范例都将使用下面的 datafile 数据库。

```
% cat datafile
northwest   NW   Joel Craig      3.0   .98   3   4
western     WE   Sharon Kelly    5.3   .97   5   23
southwest   SW   Chris Foster    2.7   .8    2   18
```

(续表)

southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17
eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

范例 6-161

```
% nawk '{if ( $8 > 15 ){ print $3 " has a high rating"}\
  else print $3 "---NOT A COMPETITOR---"}' datafile
Joel---NOT A COMPETITOR---
Sharon has a high rating
Chris has a high rating
May---NOT A COMPETITOR---
Derek has a high rating
Susan has a high rating
TJ---NOT A COMPETITOR---
Val---NOT A COMPETITOR---
Sheri---NOT A COMPETITOR---
```

说明

if 语句属于操作语句。如果表达式后的语句不止一条，就必须用花括号将它们括起来(本例中花括号可以省略，因为表达式后只有一条语句)。这个表达式的含义是：如果第 8 个字段大于 15，则打印第 3 个字段和字符串 “has a high rating”，否则就打印第 3 个字段和字符串 “---NOT A COMPETITOR---”。

范例 6-162

```
% nawk '{i=1; while(i<=NF && NR < 2){print $i; i++}}' datafile
northwest
NW
Joel
Craig
3.0
.98
3
4
```

说明

用户自定义变量 i 被赋值为 1。nawk 进入 while 循环，开始测试表达式。如果表达式值为真，就执行 print 语句，打印第 i 个字段的值。然后将变量 i 的值加 1，再次进入循环。当 i 的值大于 NF 且 NR 为 2 或更大的值时，循环表达式的值就会变为假。变量 i 的值会在开始处理下一条记录时被重新初始化。



% cat datafile

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17
eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

范例 6-163

```
% nawk '{ for( i=3 ; i <= NF && NR == 3 ; i++ ){ print $i }}' datafile
Chris
Foster
2.7
.8
2
18
```

说明

for 循环在功能上类似于 while 循环。它把初始化、测试和循环控制语句都放在一个表达式里。for 循环先为当前记录初始化一次 i 的值(i = 3)，然后测试表达式。如果 i 小于或等于 NF，并且 NR 等于 3，则执行 print 块。打印第 i 个字段的值之后，控制就转回循环表达式，将 i 的值加 1，然后再次执行测试。

范例 6-164

```
(命令行)
% cat nawk.sc4
# Awk script illustrating arrays
BEGIN{OFS="\t"}
{ list[NR] = $1 }   # The array is called list. The index is
                    # the number of the current record. The value of the
                    # first field is assigned to the array element.
END{ for( n = 1; n <= NR; n++){
        print list[n]}   # for loop is used to loop through the array
}

(命令行)
% nawk -f nawk.sc4 datafile
northwest
western
southwest
southern
southeast
eastern
```

```
northeast
north
central
```

说明

数组 list 以 NR 作为下标的值。每处理一行输入，都将其第一个字段赋值给数组 list。END 块中的 for 循环对每个数组元素执行遍历和打印操作。

范例 6-165

```
(命令行)
% cat nawk.sc5
# Awk script with special for loop
/north/{name[count++]=$3}
END{ print "The number living in a northern district: " count
    print "Their names are: "
    for ( i in name )       # Special nawk for loop is used to
        print name[i]       # iterate through the array.
}

% nawk -f nawk.sc5 datafile
The number living in a northern district: 3
Their names are:
Joel
TJ
Val
```

说明

每次在输入行中发现正则表达式 north 时，都把该行的第 3 个字段赋值给数组 name。每处理一条新记录，数组的下标 count 都被加 1，于是生成一个新的数组元素。然后，END 块采用一个特殊的 for 循环来遍历该数组。

% cat datafile

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17
eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

范例 6-166

```
(命令行)
% cat nawk.sc6
# Awk and the special for loop
```




```
{region[$1]++}   # The index is the first field of each record
END{for(item in region){
        print region[item], item
    }
}

% nawk -f nawk.sc6 datafile
1 central
1 northwest
1 western
1 southeast
1 north
1 southern
1 northeast
1 southwest
1 eastern

% nawk -f nawk.sc6 datafile3
4 Mary
2 Tom
1 Alax
1 Bob
1 Sean
```

说明

数组 region 以第一个字段作为下标，它保存的值是每个区的出现次数。然后，END 块用一个特殊的 for 循环来遍历并打印 region 数组。

习题 6 nawk 练习

(参见从本书合作站点下载的文件中名为 lab5.data 的数据库文件)

Mike Harrington:(510) 548-1278:250:100:175
Christian Dobbins:(408) 538-2358:155:90:201
Susan Dalsass:(206) 654-6279:250:60:50
Archie McNichol:(206) 548-1348:250:100:175
Jody Savage:(206) 548-1278:15:188:150
Guy Quigley:(916) 343-6410:250:100:175
Dan Savage:(406) 298-7744:450:300:275
Nancy McNeil:(206) 548-1278:250:80:75
John Goldenrod:(916) 348-4278:250:100:175
Chet Main:(510) 548-5258:50:95:135
Tom Savage:(408) 926-3456:250:168:200
Elizabeth Stachelin:(916) 440-1763:175:75:300

上面这个数据库的记录内容包括姓名、电话号码和最近 3 个月的竞选捐款数额。

请编写一个能产生如下输出的 nawk 脚本：

FIRST QUARTERLY REPORT

```
                    ***CAMPAIGN 2004 CONTRIBUTIONS***
---------------------------------------------------------------------------
  NAME              PHONE           Jan  |  Feb   |  Mar  |  Total Donated
---------------------------------------------------------------------------
Mike Harrington     (510) 548-1278  250.00  100.00  175.00   525.00
Christian Dobbins   (408) 538-2358  155.00   90.00  201.00   446.00
Susan Dalsass       (206) 654-6279  250.00   60.00   50.00   360.00
Archie McNichol     (206) 548-1348  250.00  100.00  175.00   525.00
Jody Savage         (206) 548-1278   15.00  188.00  150.00   353.00
Guy Quigley         (916) 343-6410  250.00  100.00  175.00   525.00
Dan Savage          (406) 298-7744  450.00  300.00  275.00  1025.00
Nancy McNeil        (206) 548-1278  250.00   80.00   75.00   405.00
John Goldenrod      (916) 348-4278  250.00  100.00  175.00   525.00
Chet Main           (510) 548-5258   50.00   95.00  135.00   280.00
Tom Savage          (408) 926-3456  250.00  168.00  200.00   618.00
Elizabeth Stachelin (916) 440-1763  175.00   75.00  300.00   550.00
---------------------------------------------------------------------------
                              SUMMARY
---------------------------------------------------------------------------
The campaign received a total of $6137.00 for this quarter.
The average donation for the 12 contributors was $511.42.
The highest total contribution was $1025.00 made by Dan Savage.
                    ***THANKS Dan***
The following people donated over $500 to the campaign.
They are eligible for the quarterly drawing!!
Listed are their names (sorted by last names) and phone numbers:
    John Goldenrod--(916) 348-4278
    Mike Harrington--(510) 548-1278
    Archie McNichol--(206) 548-1348
    Guy Quigley--(916) 343-6410
    Dan Savage--(406) 298-7744
    Tom Savage--(408) 926-3456
    Elizabeth Stachelin--(916) 440-1763
        Thanks to all of you for your continued support!!
```

6.25 杂项

有些数据(比如从磁带或电子表格中读取的数据)可能没有明显的字段分隔符，却有固定宽度的列。预处理这类数据时，substr 函数很管用。



6.25.1 固定字段

下面这个例子中，字段都是固定宽度的，但没有使用字段分隔符。substr 函数可以用来创建字段。

范例 6-167

```
% cat fixed
031291ax5633(408)987-0124
021589bg2435(415)866-1345
122490de1237(916)933-1234
010187ax3458(408)264-2546
092491bd9923(415)134-8900
112990bg4567(803)234-1456
070489qr3455(415)899-1426

% nawk '{printf substr($0,1,6)" ";printf substr($0,7,6)" ";\
  print substr($0,13,length)}' fixed
031291 ax5633 (408)987-0124
021589 bg2435 (415)866-1345
122490 de1237 (916)933-1234
010187 ax3458 (408)264-2546
092491 bd9923 (415)134-8900
112990 bg4567 (803)234-1456
070489 qr3455 (415)899-1426
```

说明

第 1 个字段通过从整个记录中提取子串得到，子串从记录第一个字符开始、长度为 6 个字符。接下来，打印一个空格。第 2 个字段是通过在记录中提取从位置 7 开始、长度为 6 个字符的子串得到，后跟一个空格。最后一个字段则是通过在整个记录中提取从位置 13 开始、到由行的长度所确定的位置之间的子串获得(如果未指定参数，length 函数返回当前行($0)的长度)。

空字段 如果用固定长度的字段来存储数据，就可能出现一些空字段。下面这个例子中，substr 函数被用来保存字段，而不考虑它们是否包含数据。

范例 6-168

```
1   % cat db
    xxx xxx
    xxx abc xxx
    xxx a   bbb
    xxx     xx

    % cat awkfix
    # Preserving empty fields. Field width is fixed.
    {
2   f[1]=substr($0,1,3)
3   f[2]=substr($0,5,3)
4   f[3]=substr($0,9,3)
```

```
5  line=sprintf("%-4s%-4s%-4s\n", f[1],f[2], f[3])
6  print line
   }
   % nawk -f awkfix db
   xxx xxx
   xxx abc  xxx
   xxx a    bbb
   xxx      xx
```

说明

1. 打印文件 db 的内容。这个文件中有一些空字段。
2. 数组 f 的第 1 个元素被赋值为由位置 1 开始、长度为 3 的记录的子串。
3. 数组 f 的第 2 个元素被赋值为由位置 5 开始、长度为 3 的记录的子串。
4. 数组 f 的第 3 个元素被赋值为由位置 9 开始、长度为 3 的记录的子串。
5. 用 sprintf 函数设置好数组元素的格式，然后将它们赋值给用户自定义的变量 line。
6. 打印 line 的值，可以看到结果中空字段依然被保留。

带$、逗号或其他字符的数字　下面这个例子中，价格字段中包含一个美元符号和逗号。脚本必须删掉这些字符，才能把价格加起来得出总的开销。可以通过 gsub 函数来完成这一任务。

范例 6-169

```
% cat vendor
access tech:gp237221:220:vax789:20/20:11/01/90:$1,043.00
alisa systems:bp262292:280:macintosh:new updates:06/30/91:$456.00
alisa systems:gp262345:260:vax8700:alisa talk:02/03/91:$1,598.50
apple computer:zx342567:240:macs:e-mail:06/25/90:$575.75
caci:gp262313:280:sparc station:network11.5:05/12/91:$1,250.75
datalogics:bp132455:260:microvax2:pagestation maint:07/01/90:$1,200.00
dec:zx354612:220:microvax2:vms sms:07/20/90:$1,350.00

% nawk -F: '{gsub(/\$/,"");gsub(/,/,""); cost +=$7};\
  END{print "The total is $" cost}' vendor
$7474
```

说明

第一个 gsub 函数用空字符串对美元符号(\$)进行全局替换；第二个 gsub 函数则用空串替换全部逗号。然后，将用户自定义变量 cost 与每行的第 7 个字段相加，再把每次的结果赋回给 cost，由此统计出总数。END 块打印出字符串 “The total is $”，后面跟着 cost 的值[4]。

[4] 关于如何将逗号还原到程序中的具体内容，请参见 Alfred V. Aho、Brian W. Kernighan 和 Peter J. Weinberger 的编著的 *The Awk Programming Language*(于 1998 年由波士顿的 Addison-Wesley 公司出版)。



6.25.2 多行记录

到目前为止，本书用作例子的所有数据文件中，每条记录都自成一行。而在下面这个名为 checkbook 的示例数据文件中，记录之间用空行分隔，同一记录的字段之间则用换行符分隔。要处理这个文件，就必须将记录分隔符(RS)设为空值，而把字段分隔符(FS)设为换行符。

范例 6-170

```
(输入文件)
% cat checkbook

1/1/04
#125
-695.00
Mortgage

1/1/04
#126
-56.89
PG&E
1/2/04
#127
-89.99
Safeway

1/3/04
+750.00
Paycheck

1/4/04
#128
-60.00
Visa

(脚本)
    % cat awkchecker
1   BEGIN{RS=""; FS="\n";ORS="\n\n"}
2   {print  NR, $1,$2,$3,$4}

(输出)
% nawk -f awkchecker checkbook
1 1/1/04  #125  -695.00  Mortgage

2 1/1/04  #126  -56.89  PG&E

3 1/2/04  #127  -89.99  Safeway

4 1/3/04  +750.00  Paycheck

5 1/4/04  #128  -60.00  Visa
```

说明

1. 在 BEGIN 块中，记录分隔符(RS)被赋值为空，字段分隔符(FS)被设为换行符，输出记录分隔符(ORS)则被设置为两个换行符。于是，每一行都是一个字段，且输出记录之间有两个换行符将其分隔。
2. 打印记录号，后跟记录的每个字段。

6.25.3 生成格式信函

下面这个例子改编自 *The AWK Programming Language* 中的一个程序[5]。其复杂之处在于对正在处理的数据进行记录。输入文件的名称是 data.form，它只包含数据，文件中的字段由冒号分隔。另一个文件叫做 form.letter，其内容是用于生成信函的实际格式。这个文件由 getline 函数加载进入 awk 的内存空间。格式信函的每一行都被保存在一个数组中。这个程序从 data.form 中读取数据，用实际数据替换 form.letter 中以#和@开头的特殊字符串，从而生成信函。临时变量 temp 保存了数据替换完成后，要显示的实际的行内容。这个程序可用来为 data.form 中列出的每个人生成一封个人信函。

范例 6-171

(awk 脚本)

```
% cat form.awk
# form.awk is an awk script that requires access to 2 files: The
# first file is called "form.letter." This file contains the
# format for a form letter. The awk script uses another file,
# "data.form," as its input file. This file contains the
# information that will be substituted into the form letters in
# the place of the numbers preceded by pound signs. Today's date
# is substituted in the place of "@date" in "form.letter."
1  BEGIN{ FS=":"; n=1
2  while(getline < "form.letter" > 0)
3       form[n++] = $0  # Store lines from form.letter in an array
4  "date" | getline d; split(d, today, " ")
       # Output of date is Fri Mar 2 14:35:50  PST 2004
5  thisday=today[2]". "today[3]", "today[6]
6  }
7  { for( i = 1; i < n; i++ ){
8      temp=form[i]
9      for ( j = 1; j <=NF; j++ ){
            gsub("@date", thisday, temp)
10          gsub("#" j, $j , temp )
       }
11 print temp
   }
   }
```

---

⑤ Alfred V. Aho、Brian W. Kernighan 和 Peter J. Weinberger 编著的 *The AWK Programming Language*(1988 年由波士顿的 Addison-Wesley 公司出版)，于 1988 年由 Pearson 教育公司授权贝尔电话实验室重印。




```
% cat form.letter
    The form letter, form.letter, looks like this:
*****************************************************
    Subject: Status Report for Project "#1"
    To: #2
    From: #3
    Date: @date
    This letter is to tell you, #2, that project "#1" is up to
    date.
    We expect that everything will be completed and ready for
    shipment as scheduled on #4.

    Sincerely,

    #3
*****************************************************
```

The file, data.form, is awk's **input file** containing the data that will replace the #1-4
    and the @date in form.letter.

```
% cat data.form
    Dynamo:John Stevens:Dana Smith, Mgr:4/12/2004
    Gallactius:Guy Sterling:Dana Smith, Mgr:5/18/2004
```

(命令行)

```
% nawk -f form.awk data.form
*****************************************************
Subject: Status Report for Project "Dynamo"
To: John Stevens
From: Dana Smith, Mgr
Date: Mar. 2, 2004
This letter is to tell you, John Stevens, that project
"Dynamo" is up to date.
We expect that everything will be completed and ready for
shipment as scheduled on 4/12/2001.
Sincerely,

Dana Smith, Mgr
Subject: Status Report for Project "Gallactius"
To: Guy Sterling
From: Dana Smith, Mgr
Date: Mar. 2, 2004
This letter is to tell you, Guy Sterling, that project "Gallactius"
is up to date.
We expect that everything will be completed and ready for
shipment as scheduled on 5/18/2004.

Sincerely,

Dana Smith, Mgr
```

说明

1. 在 BEGIN 块中，字段分隔符(FS)被设置为冒号。用户自定义变量 n 被设为 1。

2. 在 while 循环中，getline 函数逐行读入文件 form.letter。如果没有找到指定文件，getline 就会返回-1。读到文件末尾时，getline 返回 0。因此，只要测试 getline 的返回值是否大于 0，就能知道该函数是否从输入文件中读出一行数据。

3. 从 form.letter 中读到的每一行都被赋值给数组 form。

4. UNIX 命令 date 的输出被管道发给 getline 函数，并赋值给用户自定义变量 d。然后，split 函数用空格拆分变量 d，生成一个名为 today 的数组。

5. 把数组 today 中月、日、年的值赋给用户自定义变量 thisday。

6. BEGIN 块结束。

7. for 循环将要循环 n 次。

8. 从数组 form 中读取一行并赋值给用户自定义变量 temp。

9. 这个嵌套的 for 循环将对来自输入文件 data.form 的记录执行 NF 次循环。在变量 temp 保存的每一行中查找字符串“@date”，如果匹配，就用 gsub 函数把它替换为当天的日期(保存在 thisday 中的那个值)。

10. 如果在 temp 保存的行中发现了字符#和一个数字相连，gsub 函数就将#和这个数字替换为输入文件 data.form 中相应字段的值。例如，如果测试的对象是第一行，则#1 将替换为 Dynamo，#2 将替换为 John Stevens，#3 替换为 Dana Smith，#4 替换为 4/12/2001，以此类推。

11. 替换后，对 temp 中存储的行进行打印。

6.25.4 与 shell 交互

我们已经知道了 awk 是如何工作的，同时也了解到编写 shell 脚本时，awk 将是一个非常强大的工具。你可以在 shell 脚本中嵌入单行的 awk 命令或 awk 脚本。下面就是一个嵌入了 awk 命令的 Korn shell 程序。

范例 6-172

```
!#/bin/ksh
# This korn shell script will collect data for awk to use in
# generating form letter(s). See above.
print "Hello $LOGNAME. "
print "This report is for the month and year:"
1  cal | nawk 'NR==1{print $0}'

   if [[ -f data.form  || -f formletter? ]]
   then
      rm data.form formletter? 2> /dev/null
   fi
   integer num=1
   while true
   do
      print "Form letter #$num:"
```



```
        read project?"What is the name of the project? "
        read sender?"Who is the status report from? "
        read recipient?"Who is the status report to? "
        read due_date?"What is the completion date scheduled? "
        echo $project:$recipient:$sender:$due_date > data.form
        print -n "Do you wish to generate another form letter? "
        read answer
        if [[ "$answer" != [Yy]* ]]
        then
            break
        else
2           nawk -f form.awk  data.form  > formletter$num
        fi
        (( num+=1 ))
    done
    nawk -f form.awk data.form > formletter$num
```

说明

1. 把 UNIX 命令 cal 的输出通过管道发给 awk。打印输出的第一行，其中含有当前的月份和年份。

2. nawk 脚本 form.awk 生成的格式信函被重定向到一个 UNIX 文件。

习题 7：nawk 练习

(参见从本书合作站点下载的文件中名为 lab7.data 的数据库文件)

Mike Harrington:(510) 548-1278:250:100:175
Christian Dobbins:(408) 538-2358:155:90:201
Susan Dalsass:(206) 654-6279:250:60:50
Archie McNichol:(206) 548-1348:250:100:175
Jody Savage:(206) 548-1278:15:188:150
Guy Quigley:(916) 343-6410:250:100:175
Dan Savage:(406) 298-7744:450:300:275
Nancy McNeil:(206) 548-1278:250:80:75
John Goldenrod:(916) 348-4278:250:100:175
Chet Main:(510) 548-5258:50:95:135
Tom Savage:(408) 926-3456:250:168:200
Elizabeth Stachelin:(916) 440-1763:175:75:300

上面这个数据库所记录的内容包括姓名、电话号码和最近 3 个月的竞选捐款数额。请编写一个用户自定义函数，要求该函数能返回指定月份的人均捐款额。月份由用户在命令行输入。

6.26 awk 内置函数

本节示例，除特别声明外，都使用了下面这个重复出现过多次的 datafile 文件。

```
% cat datafile
northwest     NW    Joel Craig       3.0    .98    3    4
western       WE    Sharon Kelly     5.3    .97    5    23
southwest     SW    Chris Foster     2.7    .8     2    18
southern      SO    May Chin         5.1    .95    4    15
southeast     SE    Derek Johnson    4.0    .7     4    17
eastern       EA    Susan Beal       4.4    .84    5    20
northeast     NE    TJ Nichols       5.1    .94    3    13
north         NO    Val Shultz       4.5    .89    5    9
central       CT    Sheri Watson     5.7    .94    5    13
```

6.26.1 字符串函数

范例 6-173

```
% nawk 'NR==1{gsub(/northwest/,"southeast", $1) ;print}' datafile
southeast          NW      Joel Craig          3.0   .98    3    4
```

说明

如果这是第一条记录(NR = = 1)，就将其第 1 个字段中的正则表达式 northwest 全部替换为 southwest。当然，前提是第 1 个字段中有 northwest。

范例 6-174

```
% nawk 'NR==1{print substr($3, 1, 3)}' datafile
Joe
```

说明

如果这是第 1 条记录，就显示其第 3 个字段中从第 1 个字符开始、长度为 3 个字符的子串。显示结果是子串"Joe"。

范例 6-175

```
% nawk 'NR==1{print length($1)}' datafile
9
```

说明

如果这是第 1 条记录，就打印其第 1 个字段的长度(字符的个数)。

范例 6-176

```
% nawk 'NR==1{print index($1,"west")}' datafile
6
```



说明

如果这是第 1 条记录，就打印其第 1 个字段中首次出现子串 west 的位置。字符串 west 是从字符串 northwest 的第 6 个位置(索引)开始的。

```
% cat datafile
northwest    NW    Joel Craig       3.0    .98    3    4
western      WE    Sharon Kelly     5.3    .97    5    23
southwest    SW    Chris Foster     2.7    .8     2    18
southern     SO    `May Chin        5.1    .95    4    15
southeast    SE    Derek Johnson    4.0    .7     4    17
eastern      EA    Susan Beal       4.4    .84    5    20
northeast    NE    TJ Nichols       5.1    .94    3    13
north        NO    Val Shultz       4.5    .89    5    9
central      CT    Sheri Watson     5.7    .94    5    13
```

范例 6-177

```
% nawk '{if(match($1,/^no/)){print substr($1,RSTART,RLENGTH)}}' datafile
no
no
no
```

说明

如果 match 函数在第 1 个字段中找到了正则表达式/^no/，就返回最左端那个字符的索引位置。match 函数还将内置变量 RSTART 设置为索引位置，将变量 RLENGTH 设为匹配到的子串的长度。substr 函数返回第 1 个字段中从位置 RSTART 开始、长度为 RLENGTH 个字符的子串。

范例 6-178

```
% nawk 'BEGIN{split("10/14/04",now,"/");print now[1],now[2],now[3]}'
10 14 04
```

说明

字符串 10/14/01 被拆分成数组 now。分隔符是正斜杠。从第 1 个元素开始打印数组的所有元素。

范例 6-179 使用的是下面的 datafile2 数据库。

```
% cat datafile2
Joel Craig:northwest:NW:3.0:.98:3:4
Sharon Kelly:western:WE:5.3:.97:5:23
Chris Foster:southwest:SW:2.7:.8:2:18
May Chin:southern:SO:5.1:.95:4:15
Derek Johnson:southeast:SE:4.0:.7:4:17
```

(续表)

Susan Beal:eastern:EA:4.4:.84:5:20

TJ Nichols:northeast:NE:5.1:.94:3:13

Val Shultz:north:NO:4.5:.89:5:9

Sheri Watson:central:CT:5.7:.94:5:13

范例 6-179

```
% nawk -F: '/north/{split($1, name, " ");\
  print "First name: "name[1];\
  print "Last name: " name[2];\
  print "\n--------------------"}' datafile2

First name: Joel
Last name: Craig
--------------------
First name: TJ
Last name: Nichols
--------------------
First name: Val
last name: Shultz
--------------------
```

说明

输入字段分隔符被设为冒号(-F:)。如果记录中包含正则表达式 north，就用空格作为分隔符，将该记录的第 1 个字段拆分成数组 name。然后打印数组 name 的元素。

% cat datafile

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17
eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

范例 6-180

```
% nawk '{line=sprintf("%10.2f%5s\n",$7,$2); print line}' datafile
3.00   NW
5.00   WE
2.00   SW
```




```
4.00   SO
4.00   SE
5.00   EA
3.00   NE
5.00   NO
5.00   CT
```

说明

sprintf 函数以 printf 函数的格式规范来设置第 7 和第 2 个字段($7、$2)的格式。接着，返回经过格式化的字符串，并将其赋值给用户自定义的变量 line。然后，变量 line 被打印出来。

toupper 和 tolower 函数(仅适用于 gawk) toupper 函数返回的是一个字符串。其中，所有的小写字母均已转换成大写字母，非字母字符不发生变化。类似地，tolower 也返回字符串，但作用于将所有大写字母转换成小写字母。注意，字符串必须加双引号。

格式

```
toupper(字符串)
tolower(字符串)
```

范例 6-181

```
% awk 'BEGIN{print toupper("Linux"),tolower("BASH 2.0")}'
LINUX bash 2.0
```

6.26.2 gawk 的时间函数

gawk 提供了两个函数来获取时间和格式化时间戳：systime 和 strftime。

systime 函数 systime 函数将返回自 1970 年 1 月 1 日以来经过的时间(按秒计算)。

格式

```
systime()
```

范例 6-182

```
% awk 'BEGIN{now=systime(); print now}'
939515282
```

说明

systime 函数的返回值被赋给一个用户自定义的变量：now。这个值等于从 1970 年 1 月 1 日以来所累计的总时间(单位为秒)。

strftime 函数。 strftime 函数使用 C 库中的 strftime 函数对时间进行格式化。格式形式可以为%T %D 等(参见表 6-14)。时间戳的格式和 systime 函数返回值所采用的格式一样，如果不使用时间戳，则以当前的时间为默认时间。

表 6-14 日期和时间格式规范

日期格式	定　义
本表基于如下假设 当前日期：2004 年 10 月 17 日，星期日 当前时间：美国时间 15:26:26	
%a	简写的星期名(如 Sun)
%A	完整的星期名(如 Sunday)
%b	简写的月名(如 Oct)
%B	完整的月名(如 October)
%c	本地的日期和时间(如 Sun Oct 17 15:26:46 2004)
%d	用十进制表示的月份中的某一天(如 17)
%D	采用 10/17/04 形式表示的日期[⑥]
%e	月份中的某一天，如果只有一位数字，用空格填充[⑥]
%H	用十进制表示的 24 小时制的小时数(如 15)
%I	用十进制表示的 12 小时制的小时数(如 03)
%j	用十进制表示的从当年 1 月 1 日以来的天数(如 290)
%m	用十进制表示的月数(如 10)
%M	用十进制表示的分钟数(如 26)
%p	采用 12 小时制表示的 AM/PM 表示法(如 PM)
%S	用十进制表示的秒数(如 26)
%U	用十进制表示的一年中的周数(星期日作为一周的开始)(如 42)
%w	用十进制表示的星期数(如星期日为 0)
%W	用十进制表示的一年中的周数(星期一作为一周的开始)(如 41)
%x	本地日期(如 10/17/04)
%X	本地时间(如 15:26:26)
%y	用十进制表示的年份(采用两位十进制表示，如 04)
%Y	带世纪的年份(如 2004)
%Z	时间区(如 PDT)
%%	一个百分号字符标记(%)

格式

```
systime([format specification][,timestamp])
```

范例 6-183

```
% awk 'BEGIN{now=strftime("%D", systime()); print now}'
10/09/04
% awk 'BEGIN{now=strftime("%T"); print now}'
17:58:03
% awk 'BEGIN{now=strftime("%m/%d/%y"); print now}'
```

⑥ %D 和%e 仅在 gawk 的一些版本上可以使用。

```
10/09/04
```

说明

strftime 函数通过一个参数所给出的格式(参见表 6-14）来设置时间和日期的形式。如果以 systime 作为第 2 个参数，或者不带第 2 个参数，将使用本地的当前时间。如果带了第 2 个参数，则它必须与 systime 函数的返回值格式一致。

下一节将使用 datafile 数据库，为了方便读者阅读，这里再次给出了该数据库文件。

```
% cat datafile
northwest    NW    Joel Craig       3.0   .98   3   4
western      WE    Sharon Kelly     5.3   .97   5   23
southwest    SW    Chris Foster     2.7   .8    2   18
southern     SO    May Chin         5.1   .95   4   15
southeast    SE    Derek Johnson    4.0   .7    4   17
eastern      EA    Susan Beal       4.4   .84   5   20
northeast    NE    TJ Nichols       5.1   .94   3   13
north        NO    Val Shultz       4.5   .89   5   9
central      CT    Sheri Watson     5.7   .94   5   13
```

6.26.3 命令行参数

范例 6-184

```
% cat argvs.sc
# Testing command-line arguments with ARGV and ARGC using a for loop.

BEGIN{
    for(i=0;i < ARGC;i++)
        printf("argv[%d] is %s\n", i, ARGV[i])
        printf("The number of arguments, ARGC=%d\n", ARGC)
}

% nawk -f argvs.sc datafile
argv[0] is nawk
argv[1] is datafile
The number of arguments, ARGC=2
```

说明

BEGIN 块中包含一个 for 循环，用于处理命令行参数。ARGC 是参数的个数，ARGV 则是包含实际参数的数组。nawk 不把选项当成参数。这个例子中的有效参数只有 nawk 命令和输入文件 datafile。

```
% cat datafile
northwest    NW    Joel Craig       3.0   .98   3   4
western      WE    Sharon Kelly     5.3   .97   5   23
southwest    SW    Chris Foster     2.7   .8    2   18
southern     SO    May Chin         5.1   .95   4   15
southeast    SE    Derek Johnson    4.0   .7    4   17
```

(续表)

eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

范例 6-185

```
1   % nawk 'BEGIN{name=ARGV[1]};\
    $0 ~ name {print $3 , $4}'  "Derek" datafile
    nawk: can't open Derek
    source line number 1

2   % nawk 'BEGIN{name=ARGV[1]; delete ARGV[1]};\
    $0 ~ name {print $3, $4}'  "Derek" datafile
    Derek Johnson
```

说明

1. 在 BEGIN 块中，名字“Derek”被赋给变量 name。接下来的模式操作块中，nawk 试着将“Derek”作为输入文件打开，结果失败了。

2. 把“Derek”赋给变量 name 后，nawk 就把 ARGV[1]删除了。进入模式操作块时，nawk 没有尝试将“Derek”作为输入文件打开，而是打开了文件 datafile。

6.26.4 读输入(getline)

范例 6-186

```
% nawk 'BEGIN{ "date" | getline d; print d}' datafile
Mon Jan 15 11:24:24 PST 2004
```

说明

将 UNIX/Linux 的 date 命令通过管道传给 getline 函数，结果保存在变量 d 中并打印出来。

范例 6-187

```
% nawk 'BEGIN{ "date " | getline d; split( d, mon) ;print mon[2]}' datafile
Jan
```

说明

将 UNIX/Linux 的 date 命令通过管道传给 getline 函数，结果保存在变量 d 中。split 函数将字符串 d 拆分为数组 mon。然后，nawk 打印数组 mon 的第 2 个元素。

范例 6-188

```
% nawk 'BEGIN{ printf "Who are you looking for?" ; \
  getline name < "/dev/tty"};\
```



说明

从终端/dev/tty 读取输入，保存到数组 name 中。

范例 6-189

```
% nawk 'BEGIN{while(getline<"/etc/passwd"  >0){lc++};print lc}' datafile
16
```

说明

用 while 循环逐行遍历/etc/passwd 文件。每进入一次循环，都用 getline 函数读入一行，同时将变量 lc 加 1。退出循环后，打印 lc 的值，即打印出文件/etc/passwd 的行数。只要 getline 的返回值大于 0，即读入一行，循环就会继续。

6.26.5 控制函数

范例 6-190

```
% nawk '{if ( $5 >= 4.5) next; print $1}' datafile
northwest
southwest
southeast
eastern
north
```

说明

如果第 5 个字段大于 4.5，就读入输入文件(datafile)的下一行，并从 awk 脚本的起点开始处理(即 BEGIN 块)。否则，打印第一个字段。

% cat datafile

northwest	NW	Joel Craig	3.0	.98	3	4
western	WE	Sharon Kelly	5.3	.97	5	23
southwest	SW	Chris Foster	2.7	.8	2	18
southern	SO	May Chin	5.1	.95	4	15
southeast	SE	Derek Johnson	4.0	.7	4	17
eastern	EA	Susan Beal	4.4	.84	5	20
northeast	NE	TJ Nichols	5.1	.94	3	13
north	NO	Val Shultz	4.5	.89	5	9
central	CT	Sheri Watson	5.7	.94	5	13

范例 6-191

```
% nawk '{if ($2 ~ /S/){print ; exit 0}}' datafile
southwest     SW      Chris   Foster  2.7    .8     2      18

% echo $status (csh) or echo $? (sh or ksh)
0
```

说明

如果记录的第 2 个字段包含字母 S，就打印该记录，并且从 awk 程序退出。C shell 的变量 status 中保存了退出状态。如果是 Bourne shell 或 Korn shell，退出状态则保存在变量$?中。.

6.26.6 用户自定义函数

范例 6-192

```
(命令行)
% cat nawk.sc7
1  BEGIN{largest=0}
2  {maximum=max($5)}

3  function max ( num ) {
4      if ( num > largest){ largest=num }
     return largest
5  }
6  END{ print "The maximum is " maximum "."}

% nawk -f nawk.sc7 datafile
The maximum is 5.7.
```

说明

1. 用户自定义变量在 BEGIN 块中被初始化为 0。
2. 处理文件中的每一行时，都以$5 为参数调用函数 max，并将其返回值赋给变量 maximum。
3. 定义用户自定义函数 max，函数的语句必须括在花括号中。每次从输入文件 datafile 中读取新的记录后，脚本都会调用 max 函数。
4. 比较 num 和 largest 的值，返回其中较大的值。
5. 函数定义块结尾。
6. END 块打印 maximum 最终的值。

6.26.7 awk/gawk 命令行选项

awk 有多种命令行选项，而 gawk 有两种命令行选项的格式：以双中划线(--)和单词开头的 GNU 长格式，以及传统的以一个中划线和字母组成的 POSIX 短格式。gawk 的选项一般都是使用-W 选项或者是相应的长格式选项的。长格式选项的参数可以由“=”符号连接(中间没有空格)，也可以由下一行的命令行参数提供。gawk 带上选项--help(参见范例 6-193)就可以列出所有的 gawk 选项。参见表 6-15。

范例 6-193

```
% awk --help
Usage: awk [POSIX or GNU style options] -f progfile [--] file ...
       awk [POSIX or GNU style options] [--] 'program' file ...
POSIX options:          GNU long options:
```



```
        -f progfile         --file=progfile
        -F fs               --field-separator=fs
        -v var=val          --assign=var=val
        -m[fr] val
        -W compat           --compat
        -W copyleft         --copyleft
        -W copyright        --copyright
        -W help             --help
        -W lint             --lint
        -W lint-old         --lint-old
        -W posix            --posix
        -W re-interval      --re-interval
        -W source=program-text --source=program-text
        -W traditional      --traditional
        -W usage            --usage
        -W version          --version
Report bugs to bug-gnu-utils@prep.ai.mit.edu,
with a Cc: to arnold@gnu.ai.mit.edu
```

表 6-15 gawk 命令行选项

选　　项	含　　义
-F fs, --field-separator fs	指定输入字段分隔符，fs 可以为一个字符串也可以为正则表达式，例如：FS=":" 或 FS="[\t:]"
-v var=value, --assign var=value	将 value 赋值给用户定义的变量，var 于 awk 脚本前开始执行。可以用于 BEGIN 块
-f scriptfile, --file scriptfile	从 scriptfile 读入 awk 命令行
-mf nnn, -mr nnn	将存储器大小设定为 nnn。使用-mf 选项，可以限制 nnn 字段的最大值，使用-mr 选项，可以限定记录数量的最大值。不适用于 gawk
-W traditional, -W compat, --traditional --compat	运行于兼容模式上，使得 gawk 的行为和 awk(UNIX 版)的一致。gawk 上的所有扩展都将被忽略。两种模式行为一致。首选为--traditional
-W copyleft -W copyright --copyleft	打印版权的简略信息
-W help -W usage --help --usage	打印可用的 awk 选项，以及相应的功能简介
-W lint --lint	打印出警告，说明使用这种结构对传统的 UNIX awk 版本是不可移植的
-W lint-old, --lint-old	打印出警告，说明使用这种结构对传统的 UNIX awk 版本是不可移植的

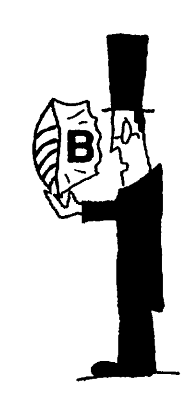
(续表)

选　　项	含　　义
-W posix --posix	打开兼容模式，此时将不能识别\x 转义序列，作为字段分隔字符的换行符(如果 FS 赋值为单个空格)、函数关键字(func)、操作符**、**=、^=、以及 fflush
-W re-interval, --re-interval	允许使用间隔正则表达式(参见 6.8.2 节的"POSIX 字符类")，即以方括号括起来的表达式，例如[[:alnum:]]
-W source program-text --source program-text	将 program-text 作为 awk 的源代码，允许 awk 命令在命令行上与-f 文件混用，例如：awk -W source '{print $1} -f cmdfile inputfile
-W version --version	打印版本和 bug 信息
--	选项处理结束标志



chapter





交互式的 Bourne shell

7.1 简介

当以交互的方式使用命令行时，shell 有一些特殊的内置变量，这些变量中包含一系列的选项。如果在选项中包含字母 i，则表示 shell 以交互方式运行。为了检测是否以交互方式运行，在提示符后键入下面的小脚本：

```
case "$-" in
*i*) echo This shell is interactive ;;
*) echo This shell is not interactive ;;
esac
```

在命令行上工作时，Bourne shell 能够提供很多使工作更加简单的特性。尽管 Bourne shell 不如本书中介绍的其他 shell 那么健壮，但是它提供文件名扩展、I/O 重定向、命令替换、函数以及其他更多的特性。在对交互式 shell 熟悉之后，就可以开始在 shell 脚本中使用这些特性了。

如果 Bourne shell 是登录 shell，它将跟在一系列进程之后启动，然后用户才能看到 shell 提示符(参见图 7-1)。

基本的启动过程如下：系统要运行的第一个进程称为 init，它的 PID 是 1。它从文件 inittab 中读取指令(System V 系统)，或者派生一个 getty 进程(BSD 系统)。这些进程打开终端端口，以提供标准输入的来源、标准输出和标准错误输出的去处，并且在屏幕上显示一个登录提示符。接下来执行的是/bin/login 程序。login 程序依次执行下面这些工作：提示用户输入口令、加密并验证用户输入的口令、设置初始环境、启动用户的登录 shell(登录 shell 是 passwd 文件的最后一项，对本章而言，就是/bin/sh)。sh 进程首先查找系统文件/etc/profile，并且执行其中的命令。然后在用户的主目录下查找名为.profile 的初始化文件。执行完文件.profile 中的命令后，屏幕上将显示默认的命令提示符，即美元符($)，然后 Bourne shell 等待用户输入命令。当然，当前大多数的系统在启动时都会提供一个带登录窗口的图形界面、一个虚拟桌面、一个用来启动 shell 终端和其他应用程序的子菜单。此时，启动的过程



会比图 7-1 所示的要复杂得多，但是一旦用户选择了终端窗口，就会出现一个提示符，用户仅键入 shell 所等待的命令即可。

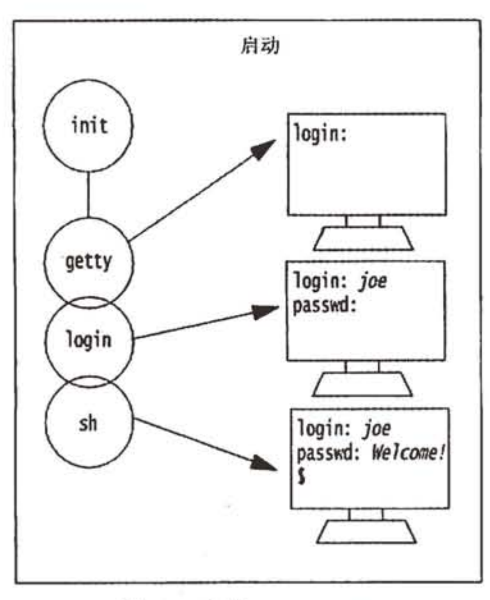


图 7-1　启动 Bourne shell

7.2　环境

进程的环境包括：变量、打开的文件、当前的工作目录、函数、资源限额、信号等。它定义了可以从一个进程继承到下一个进程的特性，以及对当前工作环境的配置。用户 shell 的配置定义在 shell 的初始化文件中。

7.2.1　初始化文件

Bourne shell 程序启动后，首先查找系统文件/etc/profile。执行完这个初始化文件中的命令，就接着执行用户主目录下的文件.profile。初始化设置的框架文件可以在/etc/skel 目录下找到(SVR4)。

/etc/profile 文件　/etc/profile 文件是一个系统级的初始化文件，由系统管理员进行设置，在用户登录时执行指定的任务。这个文件在 Bourne shell 启动时被执行。它可以被系统上的所有 Bourne shell 和 Korn shell 用户使用，通常执行诸如在邮件假脱机程序中查找新邮件、显示文件/etc/motd 中的当日信息之类的任务(学完本章之后，您会对下面这个范例有更深入的理解)。

范例 7-1

```
(Sample /etc/profile)
# The profile that all logins get before using their own .profile
1   trap " " 2 3
2   export LOGNAME PATH
```

```
3   if [ "$TERM" = " " ]
    then
        if /bin/i386
        then
            TERM=AT386   # Sets the terminal
        else
            TERM=sun
        fi
        export TERM
    fi
    # Login and -su shells get /etc/profile services.
    # -rsh is given its environment in its own .profile.
4       case "$0" in
        -sh | -ksh | -jsh )
5           if [ ! -f .hushlogin ]
            then
                /usr/sbin/quota
                # Allow the user to break the Message-Of-The-Day only.
6               trap "trap ' ' 2" 2
7               /bin/cat -s /etc/motd
                # Message of the day displayed
                trap " " 2
8               /bin/mail -E    # Checks for new mail
9               case $? in
                0)
                  echo "You have new mail. "
                ;;
                2)
                  echo "You have mail. "
                ;;
                esac
            fi
        esac
10      umask 022
11      trap 2 3
```

说明

1. trap 命令用于控制程序运行时收到的信号。如果程序在运行过程中收到信号 2(Ctrl-C)或信号 3(Ctrl-\)，将忽略它们。

2. 变量 LOGNAME 和 PATH 用于输出，以使该程序启动的子 shell 获取这两个变量的值。

3. 执行命令/bin/i386。如果命令的退出状态是 0，就将终端变量 TERM 赋值为 AT386。否则，则将变量 TERM 设为 sun。

4. 如果$0 的值，即运行文件/etc/profile 的程序名称，是一个登录的 Bourne shell、Korn shell 或作业 shell，就执行下面的命令。

5. 如果.hushlogin 文件不存在，就运行 quota 命令，如果磁盘用量超过了配额，就会显示磁盘用量的警告信息。



6. 重置 trap，让用户能够按 Control 加 C 组合键来终止当日信息的显示。

7. 显示完当日信息后，重置 trap 以忽略 Ctrl 加 C 键的作用。

8. mail 程序将检查有没有新邮件。

9. mail 程序的退出状态($?)为 0 或 2 时，将分别显示消息“You have new mail.”或“You have mail.”。

10. umask 命令用来确定文件和目录被创建时的初始权限。

11. trap 命令把对信号 2 和 3 的响应还原成默认设置，即收到 Control 加 C，或 Control 加\组合键信号后就终止进程。

.profile 文件　.profile 是用户定义的初始化文件，保存在每个用户的主目录下，用户登录时 shell 会把它执行一遍。这个文件提供了定制和修改 shell 环境的功能。环境和终端的设置通常都在这个文件里。此外，如果要启动某个窗口应用程序或数据库应用程序，也是在这里进行。我们将在本章内容逐渐深入的过程中详细讨论该文件中的设置，现在先对文件的每一行作一个简要说明。

范例 7-2

```
(Sample .profile)
1   TERM=vt102
2   HOSTNAME=`uname -n`
3   EDITOR=/usr/ucb/vi
4   PATH=/bin:/usr/ucb:/usr/bin:/usr/local:/etc:/bin:/usr/bin:.
5   PS1="$HOSTNAME $ > "
6   export TERM HOSTNAME EDITOR PATH PS1
7   stty erase ^h
8   go () { cd $1; PS1='pwd`; PS1=`basename $PS1`; }
9   trap '$HOME/.logout' EXIT
10  clear
```

说明

1. 变量 TERM 被赋值为终端的类型，即 vt102。

2. 命令 uname –n 被括在反引号中，所以 shell 将执行命令替换，即：将命令的输出(主机的名字)赋给变量 HOSTNAME。

3. 变量 EDITOR 被赋值为/usr/ucb/vi。mail 之类的程序将在指定编辑器时用到这个变量。

4. 变量 PATH 的值被设为一组目录列表，shell 查找 UNIX/Linux 程序时要搜索这些目录。例如，如果键入 ls 命令，shell 就会查找 PATH 中列出的目录，直到它在其中某个目录下找到 ls 程序为止。如果未能找到指定的程序，则 shell 会告知您这个结果。

5. 把主提示符设置为 HOSTNAME 的值(即机器名)、符号$和>。

6. 输出所列的全部变量。由这个 shell 启动的所有子程序都能识别它们。

7. stty 命令用来设置终端选项。erase 键被设置为^h，这样，当您按下退格键时，光标前面那个字符就会被删除。

8. 定义一个名为 go 的函数。此函数的目的是以一个目录名为参数，使用 cd 命令进入该目录，并将主提示符设置为当前工作目录。basename 命令则删除所有其他的路径，只保

留最后一个。这样，提示符就可以显示当前的目录。

9. trap 命令是一个信号处理命令。当您退出 shell 时(即注销时)，shell 会执行.logout 文件。

10. clear 命令清空屏幕。

7.2.2 提示符

交互式使用时，shell 会提示用户输入命令。看到提示符，就可以开始输入命令了。Bourne shell 提供了两种提示符：主提示符，即美元符($)；次提示符，即一个向右的尖括号(>)。交互运行时，shell 会显示这两个提示符。也可以改变提示符。变量 PS1 代表主提示符，其初始值被设为美元符($)。用户登录后，shell 等待用户键入命令时，屏幕上就会出现主提示符。变量 PS2 代表次提示符，其初始值是向右的尖括号。如果用户未将命令输完整就按了回车键，屏幕上就会显示次提示符。主提示符和次提示符可以重新设定。

主提示符　默认的主提示符是美元符。可以变为自己想要的提示符。通常，提示符是在用户的初始化文件.profile 中定义。

范例 7-3

```
1   $ PS1="`uname -n > `"
2   chargers >
```

说明

1. 默认的主提示符是美元符($)。这条命令把提示符 PS1 重置为主机名(uname –n)和符号>(不要把反引号和单引号搞混)。

2. 显示新的提示符。

次提示符　PS2 提示符就是次提示符。它的值显示在标准错误输出(默认为屏幕)上。如果没有输入完整的命令就按下回车符，就会出现这个提示符。

范例 7-4

```
1   $ echo "Hello
2   > there"
3   Hello
    there
4   $

5   $ PS2="----> "
6   $ echo 'Hi
7   ---->
    ---->
    ----> there'
    Hi

    there
    $
```

说明

1. 字符串("Hello)后面必须有一个对应的双引号。

2. 输入换行符后，出现了次提示符。如果不输入闭合双引号，次提示符就会继续出现。



3. 显示 echo 命令的输出。

4. 显示主提示符。

5. 重新设置次提示符。

6. 字符串('Hi)后面必须有一个对应的单引号。

7. 输入换行符后，出现了新设置的次提示符。如果不输入闭合单引号，次提示符就会继续出现。

7.2.3 搜索路径

变量 PATH 被 Bourne shell 用于定位用户在命令行键入的命令。搜索路径就保存在 PATH 中，它是一个由冒号分隔的目录列表。shell 用这些路径来查找命令，从左到右依次搜索。路径末尾的句点代表当前工作路径。如果未在 path 中列出的所有目录里找到目标命令，Bourne shell 就会向标准错误输出发送消息——filename: not found。通常推荐在文件.profile 中设置路径。

如果路径中未包含句点，则执行当前工作目录下的命令或脚本时，必须在脚本的名字前面加上./，例如./program_name，这样 shell 才能找到该程序。

范例 7-5

```
(显示 PATH)
1   $ echo $PATH
    /home/gsa12/bin:/usr/ucb:/usr/bin:/usr/local/bin:/usr/bin:
    /usr/local/bin:.

(设置 PATH)
2   $ PATH=$HOME:/usr/ucb:/usr:/usr/bin:/usr/local/bin:.
3   $ export PATH
```

说明

1. 通过回显$PATH，可以显示出变量 PATH 的值。路径包括以冒号分隔的目录列表，对路径的查找是从左到右进行的。路径末尾的句点代表用户的当前工作路径。

2. 设置路径，把以冒号分隔的目录列表赋值给 PATH 变量。

3. 输出路径，让子进程也能访问它。

7.2.4 hash 命令

hash 命令用于控制系统内部的一个哈希表，shell 将用这个表来提高命令查找的效率。有了这个内部哈希表，shell 就不必对输入的每一条命令都去搜索路径。第一次处理某条命令时，shell 通过搜索路径找到这条命令，然后将它保存在 shell 内存空间的一个表中。当再次使用同一命令时，shell 通过哈希表来查找它。这样访问命令比起搜索整个路径来要快得多。如果事先能确定要经常使用某条命令，就可以将它加到哈希表中。同样，也可以从这个表中删除不常用的命令。hash 命令的输出结果显示了 shell 通过该表找到某条命令的次数(hits)，以及查找该命令的相应开销(cost)，即要搜索多少个路径中的目录才能找到这条命令。带-r 选项的 hash 命令将会清空这个哈希表。

范例 7-6

```
1  $ hash
   hits  cost   command
   3     8      /usr/bin/date
   1     8      /usr/bin/who
   1     8      /usr/bin/ls
2  $ hash vi
   3     8      /usr/bin/date
   1     8      /usr/bin/who
   1     8      /usr/bin/ls
   0     6      /usr/ucb/vi
3  $ hash -r
```

说明

1. hash 命令显示此刻保存在内部哈希表中的命令。当用户在命令行输入这些命令时，shell 不必到列出的路径中去查找它们，从而节省时间。否则，shell 将必须到磁盘上去搜索路径。如果是一条新命令，shell 将首先在列出的路径中查找它，然后把它加入哈希表。因此，当用户再次使用这条命令时，shell 就可以在内存中找到它。

2. hash 命令可以带参数。参数名就是那些要提前存入哈希表中的命令的名称。

3. 带-r 选项的 hash 命令用来清空哈希表。

7.2.5 dot 命令

dot 命令是内置的 Bourne shell 命令。它以一个脚本名为参数。shell 将在当前的环境中执行这个脚本，也就是说，不会为它启动一个子进程。这个脚本中设置的所有参数都将成为当前 shell 环境的一部分。同样，当前 shell 设置的所有参数也都会成为该脚本的环境的一部分。dot 命令通常被用来重新执行经过修改的.profile 文件。例如，如果你在登录之后修改了.profile 中某项设置，比如变量 EDITOR 或 TERM，用 dot 命令重新执行.profile 就可以让修改生效，而不必先注销再重新登录回来。

范例 7-7

```
$ . .profile
```

说明

dot 命令在当前 shell 中执行初始化文件.profile。局部和全局变量都将在当前 shell 中重新定义。dot 命令可以免去必须先注销再重新登录回来的麻烦[①]。

7.3 命令行

用户登录成功后，shell 会显示它的主提示符，默认情况下是一个美元符。shell 其实就

① 如果把.profile 作为脚本直接运行，就会启动一个子 shell。其结果是变量只在子 shell，而不是登录 shell(即父 shell)中被设置。



是命令解释器。当以交互方式运行时，shell 从终端读取命令，并把命令行分解为若干个单词。命令行由一个或多个单词(标记)组成，以空白符(空格或制表符)来分隔，以换行符结尾，换行符则是通过按下回车键产生的。命令行的第一个词是命令，后续的词则是命令的参数。命令可以是一个 UNIX 可执行程序(比如 ls 和 pwd)，也可以是 shell 的一条内置命令(比如 cd 和 test)或某个 shell 脚本。命令(的名字)可能含有称作元字符的特殊字符，shell 分析命令行时必须解释这些元字符。如果命令行很长，且需要转到下一行继续输入，就必须先输入一个反斜杠，然后再换行，这样才能在下一行接着输入。命令行结束之前，shell 会在接下来的每一行上显示次提示符。

7.3.1 退出状态

命令或程序终止后，会向父进程返回一个退出状态。退出状态是一个 0~255 之间的整数。通常，程序退出时，如果返回的状态是 0，表示命令执行成功，如果退出状态非 0，则表示命令因某种原因而执行失败。shell 的状态变量(？)被设置为 shell 执行的上一条命令的退出状态值。程序运行结果是成功还是失败，将由编写它的程序员进行确认。

范例 7-8

```
1   $ grep "john" /etc/passwd
    john:MgVyBsZJavd16s:9496:40:John Doe:/home/falcon/john:/bin/sh
2   $ echo $?
    0
3   $ grep "nicky" /etc/passwd
4   $ echo $?
    1
5   $ grep "scott" /etc/passsswd
    grep: /etc/passsswd: No such file or directory
6   $ echo $?
    2
```

说明

1. grep 程序在文件/etc/passwd 中查找模式 John，并且成功地找到了。于是 grep 显示/etc/passwd 中匹配的行。
2. 变量“？”被设置为 grep 命令的退出状态值。值为 0 表示成功。
3. grep 程序在文件/etc/passwd 中找到用户 nicky。
4. grep 程序如果找不到模式，就会返回退出状态 1。
5. grep 因为打不开文件/etc/passsswd 而运行失败。
6. grep 如果找不到文件，就会返回退出状态 2。

7.3.2 含多条命令的命令行

一个命令行可以包括多条命令。命令之间用分号隔开，命令行以换行符终止。

范例 7-9

```
$ ls; pwd; date
```

说明

从左到右逐一地执行命令，直至遇到换行符。

命令分组 可以把多条命令合成一组，这样就能将所有命令的输出通过管道发给另一条命令，或者重定向到某个文件。

范例 7-10

```
$ ( ls ; pwd; date ) > outputfile
```

说明

每条命令的输出都被发送到文件 outputfile。圆括号内侧的空格是必需的。

7.3.3 命令的条件执行

在条件下地执行命令时，要用特殊的元字符，即双与号(&&)或双竖杠(||)来分隔两个命令串。是否执行这两个元字符右侧的命令取决于左侧命令的退出状态。

范例 7-11

```
$ cc prgm1.c -o prgm1 && prgm1
```

说明

如果第一条命令执行成功(退出状态为 0)，就执行&&后面的命令。也就是说，如果 cc 程序能成功编译 prgm1.c，就执行它生成的可执行程序 prgm1。

范例 7-12

```
$ cc prog.c 2> err || mail bob < err
```

说明

如果第一条命令执行失败(有一个不为 0 的退出状态)，就执行||后面的命令；即：如果 cc 程序未能成功编译 prgm1.c，就把报错信息写到文件 err 中，然后通过邮件将 err 发给用户 bob。

7.3.4 在后台执行的命令

通常在执行命令时，命令都在前台运行，要等到命令执行完之后，提示符才会重新出现。等待命令结束有时会不太方便。如果在命令行末尾加上一个与号(&),shell 就会立即返回 shell 提示符，同时在后台执行这条命令。于是，你不需要等待就能执行另一条命令。后台任务会在执行过程中将它产生的输出随时发送到屏幕上。因此，如果想在后台运行一条命令，不妨将它输出重定向到某个文件，或者通过管道发给某个设备(比如打印机)，这样，后台命令的输出结果就不会干扰你正在前台进行的工作。

变量$!用于保存最后一个进入后台的作业的 PID 号。

范例 7-13

```
1   $ man sh | lp&
2   [1] 1557
3   $ kill -9 $!
```



说明

1. 通过管道将 man 命令的输出(即 sh 命令的手册页)发给打印机。命令行末尾的与号把作业放入后台。

2. 屏幕上显示了两个数字：方括号里的数字说明这是第一个被放入后台的作业；第 2 个数是一个 PID，或者说该作业的进程标识号。

3. shell 提示符立刻就出现了。程序在后台运行时，shell 正等待着下一条将在前台运行的命令。变量!的值是最近那个被放入后台的作业的 PID。如果能及时取到这个值，就能赶在作业进入打印队列之前终止它。

7.4 元字符(通配符)

与 grep、sed 和 awk 相似，shell 也有元字符。粗看起来，它们似乎是一样的，但是 shell 对这些字符的解释与上述工具软件的并不完全一致。元字符(参见第 4 章)是用来代表本身含义以外的内容的特殊字符。shell 的元字符也称为通配符(wildcards)。表 7-1 列出了 Bourne shell 的元字符及其功能。

表 7-1 Bourne shell 的元字符

元 字 符	含 义
\	按字面含义解释它后面那个字符
&	在后台处理的进程
;	分隔命令
$	替换变量
?	匹配单个字符
[abc]	匹配这组字符中的一个
[!abc]	匹配这组字符以外的某个字符
*	匹配零个或多个字符
(cmds)	在子 shell 中执行命令
{cmds}	在当前 shell 中执行命令

7.5 文件名替换

确定命令行时，shell 会用元字符来缩写能够匹配某个特定字符组的文件名或路径名。如表 7-2 中所列的文件名替换元字符可以扩展为一组按字母顺序排列的文件名。将元字符扩展为文件名的过程又被称作文件名替换(filename substitution)或 globing。如果没有文件名能够跟所用的元字符匹配，shell 就会把这个元字符当作一个字面字符。

表 7-2 Bourne shell 元字符与文件名替换

元 字 符	含 义
*	匹配零个或多个字符
?	匹配单一字符
[abc]	匹配 a、b、c 这组字符中的任一个
[a-z]	匹配在 a 至 z 这个范围内的某个字符
[!a-z]	匹配不在 a 至 z 这个范围内的某个字符
\	转义或禁用后面那个元字符

7.5.1 星号

星号是一个通配符，它匹配文件名中零个或多个任意字符。

范例 7-14

```
1   $ ls *
    abc abc1 abc122 abc123 abc2 file1 file1.bak file2 file2.bak none
    nonsense nobody nothing nowhere one
2   $ ls *.bak
    file1.bak file2.bak
3   $ echo a*
    ab abc1 abc122 abc123 abc2
```

说明

1. 星号被展开为当前工作目录下的所有文件名。所有文件名都被作为参数传给 ls 命令并被显示。

2. 匹配并列出所有以零个或多个字符开头、.bak 结尾的文件名。

3. 匹配所有以 a 开头、后跟零个或多个字符的文件名，并将它们作为参数传给 echo 命令。

7.5.2 问号

问号代表文件名中的单个字符。当文件名中包含一个或多个问号时，shell 把问号替换为在文件名中匹配到的字符，从而以这种方式来完成文件名替换。

范例 7-15

```
1   $ ls
    abc  abc122  abc2  file1.bak  file2.bak  nonsense  nothing  one
    abc1  abc123  file1  file2  none  noone  nowhere
2   $ ls a?c?
    abc1 abc2
3   $ ls ??
    ?? not found
4   $ echo  abc???
    abc122 abc123
5   $ echo ??
    ??
```



说明

1. 列出当前目录下的文件。

2. 匹配并列出以 a 开头，后跟一个字符，再跟字符 c 和一个字符的文件名。

3. 如果找到正好由两个字符组成的文件名，就列出来。因为当前目录下没有名为两个字符的文件，所以这两个问号被解释为由两个字面字符“?”组成的文件名。

4. 以 abc 开头、后跟 3 个字符的文件名，由 echo 命令扩展并显示。

5. 当前目录下没有名字正好是两个字符的文件。shell 无法匹配时，就把问号按字面意义来理解。

7.5.3 方括号

方括号用于匹配包含指定字符组或字符范围内某个字符的文件名。

范例 7-16

```
1  $ ls
   abc  abc122 abc2 file1.bak file2.bak nonsense nothing
   one  abc1 abc123 file1 file2 none noone nowhere
2  $ ls abc[123]
   abc1  abc2
3  $ ls abc[1-3]
   abc1  abc2
4  $ ls [a-z][a-z][a-z]
   abc one
5  $ ls [!f-z]???
   abc1  abc2
6  $ ls abc12[23]
   abc122 abc123
```

说明

1. 列出当前目录下的所有文件。

2. 检查所有包含 4 个字符的文件名，列出以 abc 开头，后跟 1、2 或 3 的文件名。只对方括号中的一个字符进行匹配。

3. 检查所有包含 4 个字符的文件名，列出以 abc 开头，后跟一个 1~3 之间的数字的文件名。

4. 检查所有包含 3 个字符的文件名，列出由 3 个小写字母组成的文件名。

5. 列出所有包含 4 个字符、第 1 个字符不是 f~z 之间的小写字母([!f-z])，后面是 3 个任意字符(例如，???)的文件名。

6. 列出文件名以 abc12 开头，后跟 2 或 3 的文件。

7.5.4 转义元字符

如果想把元字符当作字面字符使用，可以用反斜杠来阻止 shell 解释元字符。

范例 7-17

```
1   $ ls
    abc file1 youx
2   $ echo How are you?
    How are youx
3   $ echo How are you\?
    How are you?
4   $ echo When does this line \
    > ever end\?
    When does this line ever end?
```

说明

1. 列出当前目录下的所有文件(注意 youx 这个文件)。

2. shell 对“?”执行文件名扩展。匹配当前目录下所有以 you 开头，后面有且只有一个字符的文件名，把它们替换到字符串中。文件名 youx 将被替换到字符串中，结果字符串变成了：How are youx (这并非想要的结果)。

3. 在问号前面加一个反斜杠，问号就被转义了，这样，shell 便不会再解释成通配符。

4. 在换行符前面加一个反斜杠将其转义。次提示符将一直出现直到字符串以换行符终止。转义问号“?”，不对它执行文件名替换。

7.6 变量

变量可分为两类：局部变量和环境变量。有些变量是用户创建的，其他的则是专用的 shell 变量。

7.6.1 局部变量

局部变量的值只对创建它们的 shell 可见。变量名必须以字母或下划线字符开头，后面的字符则可以是字母、0~9 的十进制数字或下划线。此外的任意字符都标志着变量名的终止。赋值时，等号两侧不能有空格。如果要将变量设为空值，就要紧接着等号按下回车键[②]。

在变量前面加一个美元符，表示要提取变量保存的值。

7.6.2 设置局部变量

局部变量的设置如范例 7-18 所示。

范例 7-18

```
1   $ round=world
    $ echo $round
    world
2   $ name="Peter Piper"
    $ echo $name
```

② 运行 set 命令时，凡是被设置为某个值或空值的变量都会被显示出来，未经设置的变量则不会被显示。



```
    Peter Piper
3   $ x=
    $ echo $x
4   $ file.bak="$HOME/junk"
    file.bak=/home/jody/ellie/junk: not found
```

说明

1. 将变量 round 赋值为 world。如果遇到变量名前面是一个美元符的情况，shell 就执行变量替换。该命令将显示变量 round 的值。

2. 变量 name 被赋值为"Peter Piper"。必须用引号来隐藏空白符，这样，shell 分析命令行时才不会将这个字符串割裂为两个词。该命令显示变量 name 的值。

3. 这条命令没有给变量 x 赋值，因此 x 被赋值为空。结果显示一个空值，即空字符串。

4. 变量名中出现句点是非法的。变量名可以使用的字符只能是数字、字母和下划线。shell 试着将这个字符串作为一条命令来执行。

局部变量的作用域　局部变量只能被创建它的 shell 识别。shell 不会将局部变量传给子 shell。双美元符是一个特殊的变量，它保存了当前 shell 的 PID。

范例 7-19

```
1   $ echo $$
    1313
2   $ round=world
    $ echo $round
    world
3   $ sh          # Start a subshell
4   $ echo $$
    1326
5   $ echo $round

6   $ exit         # Exits this shell, returns to parent shell
7   $ echo $$
    1313
8   $ echo $round
    world
```

说明

1. 双美元符变量的值是当前 shell 的 PID。本例中这个 shell 的 PID 是 1313。
2. 将局部变量 round 赋值为字符串 world，并打印该变量的值。
3. 另外启动一个 Bourne shell，这个 shell 被称为子 shell(subshell 或 child shell)。
4. 当前这个 shell 的 PID 是 1326，其父 shell 的 PID 则是 1313。
5. 变量 round 在这个 shell 中没有定义，因此打印了一个空行。
6. exit 命令终止当前进程，返回到父进程(按 Ctrl+D 组合键也能退出当前进程)。
7. 返回到父进程，并显示它的 PID。
8. 显示变量 round 的值。

设置只读变量　只读变量不能被重新定义或复位。

范例 7-20

```
1   $ name=Tom
2   $ readonly name
    $ echo $name
    Tom
3   $ unset name
    name: readonly
4   $ name=Joe
    name: readonly
```

说明

1. 局部变量 name 的值被设为 Tom。
2. 将变量 name 设为只读。
3. 不能复位只读变量。
4. 不能重新定义只读变量。

7.6.3 环境变量

环境变量可用于创建它们的 shell 和该 shell 派生的所有子 shell 和进程。按照惯例，环境变量应该用大写字母表示。环境变量就是已被输出的变量。

变量被创建时所处的 shell 被称为父 shell。如果父 shell 又启动了一个 shell，这个新的 shell 被称作子 shell。有一些环境变量，比如 HOME、LOGNAME、PATH 和 SHELL，在用户登录之前就已经被/bin/login 程序设置好了。通常情况下，环境变量被定义和保存在用户主目录下的.profile 文件里。请参见表 7-3 中列出的环境变量。

表 7-3　Bourne shell 的环境变量

环 境 变 量	值
PATH	命令的搜索路径
HOME	主目录，若未指定目录，则使用 cd 命令来指定它
IFS	内部字段分隔符，通常是空格、制表符和回车
LOGNAME	用户的登录名
MAIL	如果该参数被设置为某个邮件文件的名称，而 MAILPATH 参数未被设置，当邮件到达 MAIL 指定的文件时，shell 会通知用户
MAILCHECK	这个参数定义 shell 将隔多长时间(以秒为单位)检查一次由参数 MAILPATH 或 MAIL 指定的文件，看看是否有邮件到达。默认值是 600 秒(10 分钟)。如果将它设为 0，则 shell 每次输出主提示符之前都会去检查邮件
MAILPATH	由冒号分隔的文件名列表。如果设置了这个参数，只要有邮件到达任何一个由它指定的文件，shell 都会通知用户。每个文件名后面都可以跟一个百分号和一条消息，当文件修改时间发生变化时，shell 会打印这条消息。默认的消息是：You have mail
PWD	当前工作目录
PS1	主提示符，默认为美元符
PS2	次提示符，默认为右尖括号
SHELL	shell 启动时，会在环境中查找这个名字。shell 把默认值赋给 PATH、PS1、PS2、MAILCHECK 和 IFS。HOME 和 MAIL 由 login 程序设置



设置环境变量 如果想设置环境变量，就要在给变量赋值之后或设置变量时使用 export 命令(导出变量时，不要在变量前加美元符)。

范例 7-21

```
1   $ TERM=wyse
    $ export TERM
2   $ NAME="John Smith"
    $ export NAME
    $ echo $NAME
    John Smith
3   $ echo $$
    319             # pid number for parent shell
4   $ sh            # Start a subshell
5   $ echo $$
    340             # pid number for new shell
6   $ echo $NAME
    John Smith
7   $ NAME="April Jenner"
    $ export NAME
    $ echo $NAME
    April Jenner
8   $ exit           # Exit the subshell and go back to parent shell
9   $ echo $$
    319             # pid number for parent shell
10  $ echo $NAME
    John Smith
```

说明

1. 将变量 TERM 设置为 wyse，然后将其导出。现在，由这个 shell 启动的所有进程都将继承这个变量。

2. 定义并导出变量 NAME，让它可以被当前 shell 启动的所有子 shell 使用。

3. 打印当前 shell 的 PID 值。

4. 启动一个新的 Bourne shell。此 shell 被称为子 shell，原来那个 shell 被称为父 shell。

5. 新的 Bourne shell 的 PID 保存在变量$$中。打印这个变量的值。

6. 在父 shell 中设置的变量 NAME 被导出给这个新 shell，并显示它的值。

7. 将变量 NAME 重新设置为“April Jenner”。这个变化将导出到所有的子 shell，但不会影响父 shell。记住，导出的变量值不会向上传递给其父 shell。

8. 退出这个 Bourne 子 shell。

9. 再次打印父 shell 的 PID。

10. 变量 NANE 的值还跟原来的一样。当从父 shell 导出到子 shell 时，变量保持它们的值不变。子 shell 不可能改变父 shell 中变量的值。

7.6.4 列出已设置的变量

shell 提供了两条内置命令来打印变量的值：set 和 new。set 命令用于打印出所有的变量，局部变量和全局变量都包括在内，env 命令则只打印全局变量。

范例 7-22

```
1  $ env    (Partial list)
   LOGNAME=ellie
   TERMCAP=sun-cmd
   USER=ellie
   DISPLAY=:0.0
   SHELL=/bin/sh
   HOME=/home/jody/ellie
   TERM=sun-cmd
   LD_LIBRARY_PATH=/usr/local/OW3/lib
   PWD=/home/jody/ellie/perl

2  $ set
   DISPLAY=:0.0
   FMHOME=/usr/local/Frame-2.1X
   FONTPATH=/usr/local/OW3/lib/fonts
   HELPPATH=/usr/local/OW3/lib/locale:/usr/local/OW3/lib/help
   HOME=/home/jody/ellie
   HZ=100
   IFS=
   LANG=C
   LD_LIBRARY_PATH=/usr/local/OW3/lib
   LOGNAME=ellie
   MAILCHECK=600
   MANPATH=/usr/local/OW3/share/man:/usr/local/OW3/man:/
   usr/local/man:/usr/local/doctools/man:/usr/man
   OPTIND=1
   PATH=/home/jody/ellie:/usr/local/OW3/bin:/usr/ucb:/usr/local/
   doctools/bin:/usr/bin:/usr/local:/usr/etc:/etc:/usr/spool/
   news/bin:/home/jody/ellie/bin:/usr/lo
   PS1=$
   PS2=>
   PWD=/home/jody/ellie/kshprog/joke
   SHELL=/bin/sh
   TERM=sun-cmd
   TERMCAP=sun-cmd:te=\E[>4h:ti=\E[>4l:tc=sun:
   USER=ellie
   name=Tom
   place="San Francisco"
```

说明

1. env 命令列出所有环境变量(导出变量)。这些变量通常以大写字母命名。创建环境变量的进程将把这些变量传递给它所有的子进程。

2. 未指定选项时，set 命令将打印出所有已设置的变量，不管它是局部变量还是导出变量(被赋为空值的变量也包括在内)。

7.6.5 复位变量

只要不被设为只读，局部变量和环境变量均可以被 unset 命令复位。



范例 7-23

```
unset name; unset TERM
```

说明

unset 命令将指定变量从 shell 的内存空间删除。

7.6.6 打印变量的值：echo 命令

echo 将它的参数打印到标准输出，主要是用在 Bourne shell 和 C shell 中(Korn shell 有内置的 print 命令)。echo 命令有各种不同的版本，譬如，Berkeley(BSD)的 echo 就不同于 System V 的 echo。除非你指定一个完整的路径名，否则，你使用的将是 echo 命令的内置版本。内置版本的 echo 能够反映出你正在使用哪个版本的 UNIX。System V 的 echo 提供了很多转义序列，而 BSD 版的 echo 则没有。表 7-4 列出了 BSD 版 echo 的选项和 System V 版 echo 的转义序列。

表 7-4 BSD 版 echo 的选项和 System V 版 echo 的转义序列

选项/转义序列	含　义
BSD	
-n	删除输出结果中行尾的换行符
System V	
\b	退格
\c	打印该行，不加换行符
\f	换页
\n	换行
\r	回车
\t	制表符
\v	纵向制表符
\\	反斜杠

范例 7-24

```
1  $ echo The username is $LOGNAME.
   The username is ellie.
2  $ echo "\t\tHello there\c"   # System V
        Hello there$
3  $ echo -n "Hello there"      # BSD
   Hello there$
```

说明

1. echo 命令把它的参数显示到屏幕上。shell 会在执行 echo 命令之前先进行变量替换。

2. System V 版的 echo 命令支持的转义序列与 C 语言所用的类似。$是 shell 的提示符。

3. echo 命令的-n 选项说明使用的是 BSD 版的 echo 命令。echo 打印这行时没有加换行符。这个版本的 echo 命令不支持转义序列。

7.6.7 变量扩展修饰符

我们可以用一些专用修饰符来测试和修改变量。修饰符首先提供一个快捷的条件测试，用来检查某个变量是否已经被设置，然后根据测试结果给变量赋一个值。请参见表 7-5 中列出的 Bourne shell 变量修饰符。

表 7-5　Bourne shell 变量修饰符

修 饰 符	值
${variable:-word}	如果 variable 已被设置且值非空，就代入它的值；否则，代入 word
${variable:=word}	如果 variable 已被设置且值非空，就代入它的值；否则，将 variable 的值设为 word。且始终代入 variable 的值。而位置参量不能以这种方式赋值
${variable:+word}	如果 variable 已被设置且值非空，代入 word；否则，什么都不代入(代入空值)
${variable:?word}	如果 variable 已被设置且值非空，就代入它的值；否则，打印 word 并且从 shell 退出。如果省略了 word，就会打印信息：parameter null or not set

和冒号配合使用时，修饰符(-、=、+、?)可以检查变量是否尚未赋值或值为空。不加冒号时，值为空的变量也将被认为已设置。

范例 7-25

(将临时的默认值赋值)

```
1   $ fruit=peach
2   $ echo ${fruit:-plum}
    peach
3   $ echo ${newfruit:-apple}
    apple
4   $ echo $newfruit

5   $ echo $EDITOR        # More realistic example

6   $ echo ${EDITOR:-/bin/vi}
    /bin/vi
7   $ echo $EDITOR

8   $ name=
    $ echo ${name-Joe}

9   $ echo ${name:-Joe}
    Joe
```

说明

1. 将变量 fruit 的值设为 peach。

2. 这个专用修饰符将检查变量 fruit 是否已被设置。如果 fruit 义被设置，就打印它。否则，用 plum 替换 fruit，并打印这个值。

3. 变量 newfruit 未曾被设置。值 apple 将暂时替换 newfruit。



4. 上一行的设置是暂时的，因此，变量 newfruit 仍未被设置。

5. 环境变量 EDITOR 尚未被设置。

6. 修饰符:-将 EDITOR 替换为/bin/vi。

7. EDITOR 未曾被设置过，因此什么都不会打印。

8. 变量 name 被设为空值。因为修饰符前面没有冒号，变量即使为空也被认为是设置过的，所以没有把新的值 Joe 赋给 name。

9. 冒号使得修饰符检查变量是否未设置或为空。只要是这两种情况之一，就用值 Joe 替换 name。

范例 7-26

（为参数默认值赋值）

```
1   $ name=
2   $ echo ${name:=Patty}
    Patty
3   $ echo $name
    Patty
4   $ echo ${EDITOR:=/bin/vi}
    /bin/vi
5   $ echo $EDITOR
    /bin/vi
```

说明

1. 赋给变量 name 一个空值。

2. 专用修饰符“:=”将检查变量 name 是否尚未被设置。如果已经被设置过了，就不会对其修改。如果尚未设置或值为空，就将等号右边的值赋给它。由于之前已将变量 name 设置为空，所以现在要把 Patty 赋给它。这个设置是永久的。

3. 变量 name 的值还是 Patty。

4. 将变量 EDITOR 的值设置为/bin/vi。

5. 显示变量 EDITOR 的值。

范例 7-27

（为临时替换值赋值）

```
1   $ foo=grapes
2   $ echo ${foo:+pears}
    pears
3   $ echo $foo
    grapes
    $
```

说明

1. 将变量 foo 的值设置为 grapes。

2. 专用修饰符“:=”将检查变量 foo 是否已被设置。如果已经被设置过，就用 pears 暂时替换 foo 的值。否则，返回空。

3. 变量 foo 中显示的还是原来的值。

范例 7-28

(基于默认值创建错误消息)

```
1   $ echo ${namex:?"namex is undefined"}
    namex: namex is undefined
2   $ echo ${y?}
    y: parameter null or not set
```

说明

1. 修饰赋 “:?” 检查变量是否已被设置。如果尚未设置该变量，就把问号右边的信息打印在标准错误输出上，变量名之后。如果此时是在执行脚本，就退出脚本。

2. 如果问号后面没有提供报错信息，shell 就向标准错误输出发送默认的消息。

7.6.8 位置参数

通常，位置参数(positional parameter)，也就是特定内置变量常被 shell 脚本用来从命令行接收参数，或者被函数用来保存传给它的参数。请参见表 7-6。这些变量之所以被称为位置参数，是因为 shell 用它们在命令行上的位置来指代它们。Bourne shell 最多允许使用 9 个位置参数。shell 脚本的名称被保留在变量$0 中。可以用 set 命令来设置或复位这些位置参数。

表 7-6 Bourne Shell 位置参数

位 置 参 数	含 义
$0	引用当前 shell 脚本的名称
$1-$9	代表第 1 到第 9 个位置参数
$#	位置参数的个数
$*	所有的位置参数
$@	除了双引号引用的情况，含义与$*相同
"$*"	其值为"$1 $2 $3"
"$@"	其值为"$1" "$2" "$3"

范例 7-29

```
1   $ set tim bill ann fred
    $ echo $*       # Prints all the positional parameters.
    tim bill ann fred
2   $ echo $1       # Prints the first positional parameter.
    tim
3   $ echo $2 $3    # Prints the second and third positional parameters.
    bill ann
4   $ echo $#       # Prints the total number of positional parameters.
    4
5   $ set a b c d e f g h i j k l m
    $ echo  $10     # Prints the first positional parameter followed by a zero.
```



```
   a0
6  $ echo $*
   a b c d e f g h i j k l m
7  $ set file1 file2 file3
   $ echo \$$#
   $3
8  $ eval echo \$$#
   file3
```

说明

1. set 命令给位置参数赋值。专用变量$*包含所有的位置参数设置。

2. 显示第 1 个位置参数的值：tim。

3. 显示第 2 个和第 3 个位置参数的值，即 bill 和 ann。

4. 专用变量$#的值是当前已设置的位置参数的个数。

5. set 命令复位所有的位置参数。原来的位置参数集被破坏。位置参数的个数不能超过 9。这条命令的运行结果是打印第一个位置参数，后面跟数字 0。

6. $*能够用来打印出所有参数，即使超过 9 个也不会有问题。

7. 把位置参数重置为 file1、file2 和 file3。其中，美元符被转义，$#指的是参数的个数。echo 命令结果显示为$3。

8. 执行命令之前，eval 命令对命令行进行第二遍解析。第一遍解析时，shell 把\$$#替换为$3。第二遍解析时，shell 又将$3 替换为它的值，即 file3。

7.6.9 其他特殊变量

shell 有一些由单个字符构成的特殊变量。在字符前面加上美元符就能访问变量中保存的值。请参见表 7-7。

表 7-7 Bourne Shell 的特殊变量

变　量	含　义
$	当前 shell 的 PID
-	当前的 sh 选项设置
?	shell 执行的上一条命令的退出状态值
!	最近一个进入后台的作业的 PID

范例 7-30

```
1  $ echo The pid of this shell is $$
   The pid of this shell is 4725
2  $ echo The options for this shell are $-
   The options for this shell are s
3  $ grep dodo /etc/passwd
   $ echo $?
   1
4  $ sleep 25&
   4736
```

```
$ echo $!
4736
```

说明

1. 变量“$”保存着当前进程的 PID 值。

2. 变量“-”列出当前这个交互式 Bourne shell 的所有选项。

3. grep 命令在文件/etc/passwd 中查找字符串 dodo。变量? 保存了上一条被执行的命令的退出状态值。由于 grep 返回的值是 1，因此可以断定 grep 的查找失败了。退出状态为 0 才代表成功退出。

4. 变量! 保存着上一条被放入后台的命令的 PID 号。sleep 命令后面的“&”用来把命令放到后台。

7.7 引用

引用被用来保护特殊元字符，使其不被解释。引用有 3 种方式：反斜杠、单引号和双引号。表 7-8 中列出的字符对 shell 而言都是特殊的，必须被引用。

表 7-8 需要引用的特殊元字符

元 字 符	含 义
;	命令分隔符
&	后台处理标识
()	命令组；创建子 shell
{}	命令组；不创建子 shell
|	管道
<	输入重定向
>	输出重定向
换行符	命令终止
空格/制表符	单词分隔符
$	变量替换字符
* [] ?	用于文件名扩展的 shell 元字符

单引号和双引号都必须成对出现。单引号可以保护特殊元字符(如$、*、？、|、>和<)不被解释。双引号也能保护特殊元字符不被解释，但它允许处理变量替换字符(美元符)和命令替换字符(反引号)。单引号可以用在双引号内，双引号也可以用在单引号内。

Bourne shell 不直接显示是否存在不匹配的引号。在交互式运行状态下，如果引号不匹配，就会出现次提示符。运行 shell 脚本时，shell 先扫描文件，如果存在引号不匹配，shell 会试着找到下一个引号来匹配它。如果这种尝试失败，程序就会异常终止，并将显示这样一条信息：'end of file' unexpected。即使是顶尖的 shell 程序员，也会遇到引用带来的麻烦。



关于 shell 的引用规则，请参见第 15 章 4.2 节中的有关内容。

7.7.1 反斜杠

反斜杠用于引用(或转义)单个字符，使其免受解释。单引号里的反斜杠可以不被解释。如果是在双括号里，反斜杠将保护美元符($)、反引号(``)和反斜杠不被解释。

范例 7-31

```
1   $ echo Where are you going\?
    Where are you going?
2   $ echo Start on this line and \
    > go to the next line.
    Start on this line and go to the next line.
3   $ echo \\
    \
4   $ echo '\\'
    \\
5   $ echo '\$5.00'
    \$5.00
6   $ echo  "\$5.00"
    $5.00
```

说明

1. 反斜杠用于阻止 shell 对问号执行文件名替换。
2. 反斜杠用于转义换行符，以使下一行成为当前行的延续。
3. 反斜杠本身也是特殊字符，因此它阻止 shell 解释跟在它后面的那个反斜杠。
4. 括在单引号里的反斜杠不会被解释。
5. 单引号里的所有字符都被当成字面字符。此处的反斜杠没有任何特殊作用。
6. 括在双引号里时，反斜杠用来防止美元符被解释为变量替换。

7.7.2 单引号

单引号必须成对出现。它们能保护其中的所有元字符不被解释。要打印单引号，就必须用双引号把它括起来，或者用反斜杠转义它。

范例 7-32

```
1   $ echo 'hi there
    > how are you?
    > When will this end?
    > When the quote is matched
    > oh'
    hi there
    how are you?
    When will this end?
    When the quote is matched
    oh
2   $ echo 'Don\'t  you need $5.00?'
    Don't you need $5.00?
```

```
3   $ echo 'Mother yelled, "Time to eat!"'
    Mother yelled, "Time to eat!"
```

说明

1. 单引号没能在这一行内匹配，所以 Bourne shell 显示一个次提示符，等着引号被匹配。

2. 单引号保护了其中的所有元字符不被解释。Don't 中的撇号被转义了，否则它就会去匹配第一个单引号，导致字符串末尾的那个单引号配不成对。这个例子中的“$”和“？”都被当成字面字符，不会被 shell 解释。

3. 单引号保护字符串中的双引号不被 shell 解释。

7.7.3 双引号

双引号必须成对出现。双引号允许对它所括的内容进行变量替换和命令替换，同时保护其他的特殊字符不被 shell 解释。

范例 7-33

```
1   $ name=Jody
2   $ echo "Hi $name, I'm glad to meet you!"
    Hi Jody, I'm glad to meet you!
3   $ echo "Hey $name, the time is `date`"
    Hey Jody, the time is Wed Oct 13 14:04:11 PST 2004
```

说明

1. 将变量 name 赋值为字符串 Jody。

2. 字符串两端的双引号将保护所有的特殊元字符不被 shell 解释，$name 中的$是个例外。本例中双引号里执行了变量替换。

3. 变量替换和命令替换都可以在双引号中执行。本例中，变量 name 被替换，反引号中的 date 命令也已执行并替换。

7.8 命令替换

当命令包含反引号中时，该命令将被执行并返回输出结果。这个过程称为命令替换。命令替换用于将命令的输出结果赋给一个变量，或将命令的输出结果代入字符串。这 3 种 shell 都使用反引号来执行命令替换[3]。

范例 7-34

```
1   $ name=`nawk -F:  '{print $1}' database`
    $ echo $name
    Ebenezer Scrooge
2   $ set `date`
3   $ echo $*
```

[3] Korn shell 将反斜杠用于命令替换是为了向上兼容，它还有另一种替换方法。



```
   Fri Oct 22 09:35:21 PDT 2004
4  $ echo $2 $6
   Oct 2004
```

说明

1. 执行括在反引号内的 nawk 命令，将它的输出作为一个字符串赋给变量 name，然后显示它。

2. set 命令将 date 命令的输出赋给位置参数。用空白符分隔出单个词，然后分别赋给相应的位置参数。

3. 变量$*保存了所有的位置参数。date 命令的输出结果被保存在变量$*中。空白符用于分隔各个参数。

4. 打印第 2 个和第 6 个参数。

7.9 函数入门

虽然 Bourne shell 没有提供别名(alias)来简化命令，但它的确支持函数(SVR2 将函数引入 shell)。函数的作用是可以通过名称来执行一组命令。函数与脚本类似，只是效率更高。函数一经定义，就成了 shell 内存映像的一部分，因此，调用函数时，shell 不必像使用(脚本)文件那样取读磁盘。函数常常被用来提高脚本的模块化程度(本章的编程部分将讨论相关内容)。函数定义之后，可以被重复调用。它通常定义在用户的初始化文件.profile 中。函数必须在调用之前定义，而且无法导出。

7.9.1 定义函数

函数名后面跟有一对空的圆括号。函数的定义由花括号中的一组命令构成，命令之间以分号分隔。最后那条命令必须以分号结尾。花括号两侧的空格是必需的。

格式

```
函数名() { 命令; 命令; }
```

范例 7-35

```
1  $ greet () { echo "Hello $LOGNAME, today is `date`; }
2  $ greet
   Hello ellie, today is Thu Oct 21 19:56:31 PDT 2004
```

说明

1. 该函数名为 greet。
2. 调用函数 greet 时，shell 执行了花括号里的命令。

范例 7-36

```
1  $ fun () { pwd; ls; date; }
2  $ fun
   /home/jody/ellie/prac
```

```
    abc      abc123   file1.bak  none      nothing   tmp
    abc1     abc2     file2      nonsense  nowhere   touch
    abc122   file1    file2.bak  noone     one
    Sat Feb 21 11:15:48 PST 2004
3   $ welcome () { echo "Hi $1 and $2"; }
4   $ welcome tom joe
    Hi tom and joe
5   $ set jane nina lizzy
6   $ echo $*
    jane nina lizzy
7   $ welcome tom joe
    hi tom and joe
8   $ echo  $1 $2
    jane nina
```

说明

1. 命名并定义函数 fun。函数名后面跟了一组由花括号括着的命令。命令之间用分号分隔。第一个花括号后面必须有一个空格，否则会出现语法错误。函数必须在被调用之前被定义。

2. 函数被调用时，其行为与脚本中的一样。函数定义中的每条命令都被依次执行。

3. 函数 welcome 使用了两个位置参数。将参数传给函数后，它们的值就被赋给位置参数。

4. 函数的两个参数 tom 和 joe 被分别赋给$1 和$2。函数的位置参数为函数私有，不会对函数外面那些位置参数产生影响。

5. 在命令行设置位置参数。这些变量与函数中设置的那些位置参数没有任何关系。

6. $*显示当前已设置的位置参数的值。

7. 调用函数 welcome。赋给位置参数的值是 tom 和 joe。

8. 在命令行设置的位置参数没有被函数中设置的位置参数影响。

7.9.2 列出和复位函数

使用 set 命令可以列出函数及其定义。函数及其定义将和输出变量、局部变量一起显示在 set 的输出结果中。函数及其定义可以用 unset 命令来复位。

7.10 标准 I/O 和重定向

shell 启动时继承了 3 个文件：stdin、stdout 和 stderr。标准输入通常来自键盘。标准输出和标准错误输出通常被发往屏幕。有些时候，需要从文件读取输入，或者将输出结果和报错信息写入文件。这些都可以通过 I/O 重定向来实现。请参见表 7-9 中列出的重定向操作符。



表 7-9 重定向操作符

重定向操作符	功　能
<	重定向输入
>	重定向输出
>>	追加输出
2>	重定向标准错误输出
1>&2	将输出重定向到标准错误输出的去处
2>&1	将标准错误输出重定向到输出的去处

范例 7-37

```
1  $ tr '[A-Z]' '[a-z]' < myfile    # Redirect  input
2  $ ls > lsfile                     # Redirect output
   $ cat lsfile
   dir1
   dir2
   file1
   file2
   file3
3  $ date >> lsfile                  # Redirect and append otuput
   $ cat lsfile
   dir1
   dir2
   file1
   file2
   file3
   Mon Sept 20 12:57:22 PDT 2004
4  $ cc prog.c 2> errfile           # Redirect error
5  $ find . -name \*.c -print > foundit 2> /dev/null
                                    # Redirect output to foundit,
                                    # and error to /dev/null
6  $ find . -name \*.c -print > foundit 2>&1
                                    # Redirect output and send standard
                                    # error to where output is going
7  $ echo "File needs an argument" 1>&2
                                    # Send standard output to error
```

说明

1. UNIX 命令 tr 的标准输入被重定向为来自文件 myfile，而不是从键盘获取。显示文件 myfile 的内容，其中，所有的大写字母被转换为小写字母。

2. ls 命令将它的输出重定向到文件 lsfile，不再把输出结果发往屏幕。

3. date 命令的输出结果因重定向被追加到文件 lsfile 中。

4. 编译文件 prog.c。如果编译失败，标准错误输出被重定向到文件 errfile。现在您可以利用这个错误信息文件去请教身边的高手，他可以根据文件给出分析。

5. find 命令开始在当前工作目录下查找以.c 结尾的文件名，将找到的文件名打印到文

件 foundit 中。find 命令输出的错误信息则被发往/dev/null。

6. find 命令开始在当前工作目录下查找以.c 结尾的文件名，将找到的文件名打印到文件 foundit 中。find 命令输出的错误信息也写进 foundit。

7. echo 命令将它的信息发往标准错误输出。该命令的标准输出也一同被发往标准错误输出中。

7.10.1　exec 命令和重定向

exec 命令能够在不启动新进程的前提下，将当前正在运行的程序替换为一个新的程序。使用 exec 命令，无需创建子 shell 就能改变标准输入和输出(参见表 7-10)。用 exec 打开文件后，每条 read 命令都会将文件指针移到文件的下一行，直到文件末尾。如果要再次从头开始读文件，就必须先关闭文件再打开。但是，如果使用 cat 和 sort 这类 UNIX 工具，操作系统会在命令结束后自动关闭文件。关于如何在脚本中使用 exec 命令的示例，请参见 8.6 节“循环控制命令”。

表 7-10　exec 命令

exec 命 令	功　　能
exec ls	ls 将顶替 shell 运行。ls 运行结束后，它启动时所在的 shell 不再运行
exec < filea	打开文件 filea，用于读取标准输入
exec >filex	打开文件 filex，用于写入标准输出
exec 3<datfile	打开文件 datfile，将其作为文件描述符 3，用于读取输入
sort <&3	将文件 datfile 排序
exec 4>newfile	打开文件 newfile，将其作为文件描述符 4，用于写入输出
ls >&4	将 ls 的输出结果重定向到 newfile
exec 5<&4	使文件描述符 5 成为文件描述符 4 的一个副本
exec 3<$-	关闭文件描述符 3

范例 7-38

```
1   $ exec date
    Sun Oct 17 10:07:34  PDT 2004
     <Login prompt appears if you are in your login shell >
2   $ exec > temp
    $ ls
    $ pwd
    $ echo Hello
3   $ exec > /dev/tty
4   $ echo Hello
    Hello
```

说明

1. exec 命令在当前 shell 中执行 date 命令(不另外派生一个子 shell)。由于 date 命令是顶替当前 shell 执行，所以，当 date 命令退出后，shell 便终止了。如果这个 Bourne shell 是



从另一个C shell中启动的，结果会从Bourne shell退出，屏幕上将出现C shell的提示符。如果是在登录shell中尝试这个例子，结果将是退出系统。如果在shell窗口中交互式运行这个例子，结果则是窗口被关闭。

2. exec命令打开文件temp作为当前shell的标准输出。此后，ls、pwd和echo的输出结果将不再发往屏幕，而是写入文件temp。

3. exec命令将标准输出重新定向到终端。现在，输出结果将发到屏幕，就如第4行显示的那样。

4. 标准输出被重定向到终端(/dev/tty)。

范例7-39

```
1   $ cat doit
    pwd
    echo hello
    date
2   $ exec < doit
    /home/jody/ellie/shell
    hello
    Sun Oct 17 10:07:34  PDT 2004
3   %
```

说明

1. 显示文件doit的内容。

2. 命令exec将文件doit作为标准输入。shell将从该文件而不是键盘读取输入。exec执行文件doit中的命令，顶替当前shell来执行。当最后一条命令退出时，shell也随之退出。参见图7-2。

3. exec命令完成后，Bourne shell退出，屏幕上出现了C shell的提示符，这个C shell是刚才那个Bourne shell的父shell。假如之前是在登录shell中，那么，当exec结束时，用户将被注销。如果是在窗口中，窗口将会关闭。

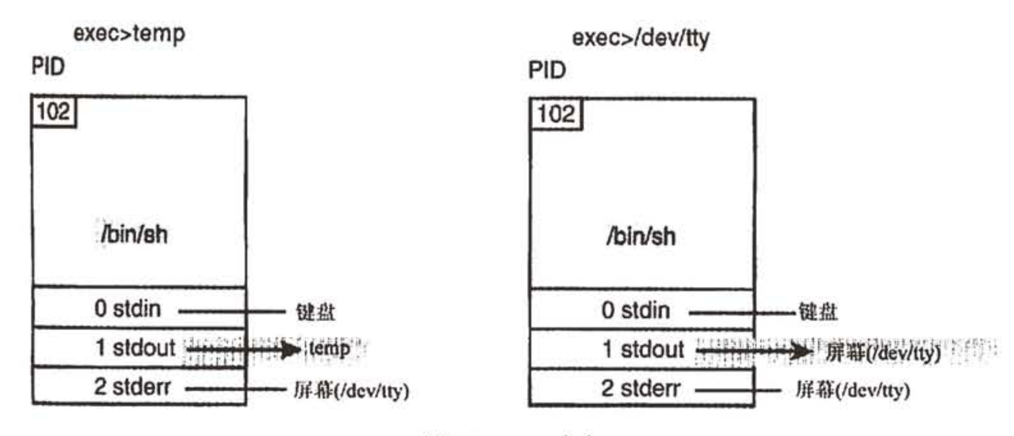


图7-2　exec命令

范例 7-40

```
1   $ exec 3> filex
2   $ who >& 3
3   $ date >& 3
4   $ exec 3>&-
5   $ exec 3< filex
6   $ cat <&3
    ellie   console   Oct 7    09:53
    ellie   ttyp0     Oct 7    09:54
    ellie   ttyp1     Oct 7    09:54
    ellie   ttyp2     Oct 11   15:42
    Sun Oct 17 13:31:31  PDT 2004
7   $ exec 3<&-
8   $ date >& 3
    Sun Oct 17 13:41:14  PDT 2004
```

说明

1. 将文件描述符 3(fd 3)指定给文件 filex，然后打开它，并将输出重定向到该文件中。参见图 7-3。

2. 将 who 命令的输出结果发往 fd 3，即文件 filex。

3. 将 date 命令的输出结果发往 fd 3。由于文件 filex 已被打开，所以结果被追加到 filex 的末尾。

4. 关闭 fd 3。

5. exec 命令打开 fd 3 用于读取输入。输入将被重定向到 filex。

6. cat 程序从 fd 3 读取输入，fd 3 已被指定给文件 filex。

7. exec 命令关闭 fd 3(实际上，一读到文件末尾，操作系统就会关闭这个文件)。

8. 试着将 date 命令的输出写到 fd 3，结果却出现在屏幕上，因为文件描述符 3 已经被关闭了。

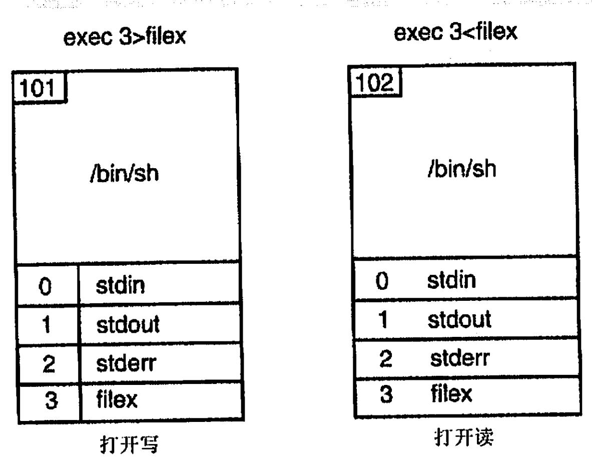


图 7-3　exec 和文件描述符



7.11 管道

管道(pipe)用于将管道符左侧命令的输出发送到管道符右侧命令的输入。一条管道线(pipeline)可以包含不止一个管道。

下面例子中这 3 条命令的目的是计算当前登录用户(who)的数目，将 who 命令的输出结果保存到一个文件(tmp)，用 wc –l 来计算文件 tmp 中的行数，然后删除文件 tmp(即已经算出了当前的登录用户数)。参见图 7-4 和图 7-5。

范例 7-41

```
1    $ who > tmp
2    $ wc -l tmp
     4 tmp
3    $ rm tmp

# Using a pipe saves disk space and time.
4    $ who | wc -l
     4
5    $ du . | sort -n | sed -n '$p'
     72388   /home/jody/ellie
```

说明

1. 将 who 命令的输出重定向到文件 tmp。
2. 命令 wc –l 用来显示文件 tmp 的行数。
3. 删除文件 tmp。
4. 使用管道功能，可以用一个步骤完成前面这 3 个步骤的全部工作。who 命令的输出被发送到内核中某个未知的缓冲区，wc –l 命令读取这个缓冲区，然后将它的输出发到屏幕上。
5. du 命令的输出(即每个目录占用磁盘块的数目)经管道发给 sort 命令，并按数值大小进行排序。之后，排序的结果又经管道发到 sed 命令，sed 命令则打印出所收到的输出的最后一行内容。

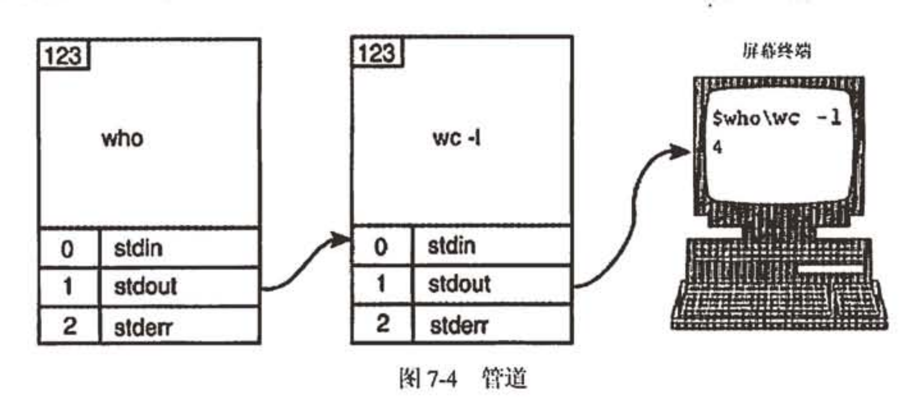


图 7-4 管道

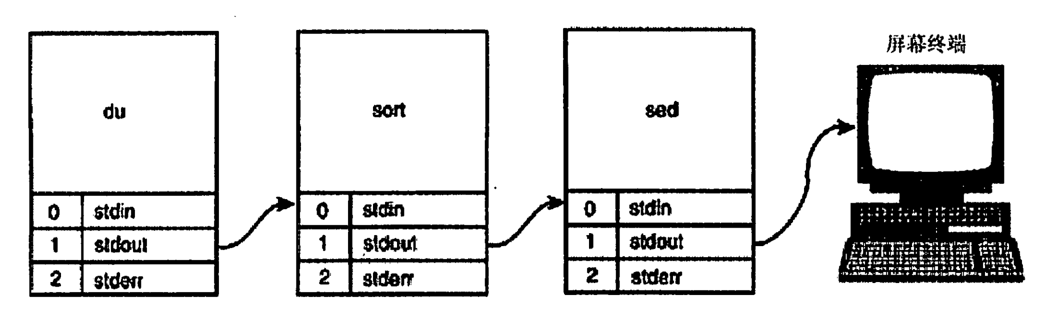


图 7-5 多重管道(容器)

7.12 here 文档与重定向输入

here 文档为需要输入数据的程序(如 mail、sort 或 cat)接收内置文本，直至遇到用户自定义的终止符。here 文档经常被 shell 脚本用来生成菜单。用来接收输入的命令后面跟着一个<<符号，<<符号后面是一个用户定义的词或符号，然后是换行符。接下来的文本行就是将要作为输入行发送给程序的正文内容。当用户定义的那个词或符号出现在某一行最左端(前后都不能有空格)，并且独占该行时，输入就终止了。这个词等同于 Ctrl+D 组合键的作用，用于通知程序停止读取输入。

范例 7-42

```
1   $ sort         # Sort waits for user input if it does get a file
    pears
    apples
    bananas
    ^D (Ctrl-D)     # User presses Ctrl-D to stop input to sort

    (Output)
    apples
    bananas
    pears
    $
2   $ sort <<DONE      # DONE is a user-defined terminator
    > pears
    > apples           # The user input is called the "document"
    > bananas
3   DONE               # Marks end of here document
                       # Here is where input stops
    (Output)
4   apples
    bananas
    pears
5   $
```

说明

1. 若 UNIX/Linux 的 sort 命令未带有文件名参数，则会等待用户的手工输入。用户键



入了 3 行字符后，按下了 Ctrl+D 组合键，即所示的键符序列。这样，就停止了向程序输入字符(注意，不要混淆 Ctrl+D 和 Ctrl+C 的用法。Ctrl+C 用于终止程序且不输出任何结果)。当用户按下 Ctrl+D 组合键后，sort 命令会将排序后的结果输出显示到屏幕上。

2. here 文档将提供 sort 命令的输入。<<符号后面跟的是用户定义的终止符(一个单词或符号)。次提示符出现，然后就都是作为 sort 命令输入内容的文本行。

3. DONE 终止符标记了 here 文档的末尾。注意，DONE 的前后不能出现空格。终止符可以取代 Ctrl+D 组合键的操作。这样，sort 命令就不会进一步接收输入，而将结果输出并显示到屏幕上。

4. 显示 sort 程序的输出。

5. shell 命令提示符重新出现。

如果定义终止符时，它前面的运算符是<<-，则输入的最后一行上，终止符前面可以出现制表符，但只能是制表符。匹配用户定义的终止词或终止符号必须做到一模一样。下面通过在命令行演示如何使用 here 文档来说明它的语法。其实，在脚本中的 here 文档要实用得多。

范例 7-43

```
1   $ cat << FINISH          <---  # FINSH is a user-defined terminator
2   > Hello there $LOGNAME      |
3   > The time is 'date'        |
    > I can't wait to see you!! |
4   > FINISH                 <---  # terminator matches first FINSH on line 1.
5   Hello there ellie
    The time is Wed April 22 19:42:16 PST 2004
    I can't wait to see you!!!
6   $
```

说明

1. UNIX 的 cat 程序将一直接收输入，直到出现自成一行的 FINISH 为止。

2. 出现次提示符，后续的文本将作为 cat 命令的输入。变量替换在 here 文档中执行。

3. 命令替换`date`在 here 文档中执行。

4. 用户定义的终止符 FINISH 标识 cat 程序输入的结束。它必需自成一行，且前后不能有空格。

5. 显示 cat 程序的输出。

6. shell 提示符重新出现。

范例 7-44

```
1   $ cat <<- DONE
    >    Hello there
    >    What's up?
    >    Bye now The time is `date`.
2   >    DONE
    Hello there
    What's up?
    Bye now The time is Thu Oct 28 19:48:23 PST 2004.
    $
```

说明

1. cat 程序接收输入，直到 DONE 出现在一行中。“<<-” 操作符表示输入和最后的终止符前可以出现一个或多个制表符(tab)。

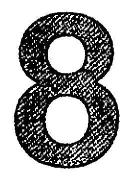
2. 匹配的终止符 DONE，其前面有一个制表符(tab)，cat 程序的输出显示在屏幕上。

小结

现在，我们已经完成了对交互式 shell 的讨论，下面的章节将讨论如何将这些特征添加进脚本中，以使程序自动完成重复的任务、产生信息、监视进程等。如果您并非一名专业的程序员，那么编写 shell 脚本可能是您学习编写程序的一个好的起始点，它将带给您更多的乐趣和直观的体验。



chapter





Bourne shell 编程

8.1 简介

shell 编程通常称为 shell 脚本(script)，我们已经学过如何使用交互式的 shell，现在就开始根据已经学过的内容来进行脚本编程，并进一步学习本章将要介绍的程序结构。不管您是系统管理员、程序员还是用户，学习脚本都是很有用的。shell 脚本可以自动执行多个例程任务，从而使工作变得更轻松，并且 shell 脚本的编写和使用都是一件很有意思的事情。

创建 shell 脚本的步骤

shell 脚本通常是在编辑器中编写的。脚本由命令和注释组成。注释跟在#号后面，包含对要执行的操作所提供的注解。

第一行 位于脚本左上角的第一行会指出要用哪个程序来执行脚本中的行。这一行通常写成：

```
#!/bin/sh
```

#!被称为幻数，内核根据它来确定该用哪个程序来解释脚本中的行。这一行必须位于脚本顶端第一行。

注释 注释是跟在#号后的行。注释可以自成一行，也可以跟在脚本命令后面与命令共处一行。注释被用来对脚本作注解。有时候，如果没有注释，就很难理解脚本究竟可以用来做什么。注释很重要，但是脚本中却经常缺少注释，甚至根本就没有注释。我们要尽量养成写注释的习惯，不光为别人着想，也方便自己。因为过了两天你可能就无法清晰地记起当时要做些什么。

可执行语句与 Bourne shell 结构 Bourne shell 程序由 UNIX 命令、Bourne shell 命令、编程结构和注释组合而成。

执行脚本 创建文件时，并不会授予文件的执行权限。如果要运行脚本，就必须指定它的执行权限。可以用 chmod 命令来设置脚本的执行权限。



范例 8-1

```
1  $ chmod +x myscript
2  $ ls -lF myscript
   -rwxr-xr-x    1   ellie   0 Jul   13:00 myscript*
```

说明

1. 用 chmod 命令打开文件属主、属组和其他用户的执行权限。

2. ls 命令的输出表明所有用户都拥有文件 myscript 的执行权限。文件名尾部的星号也说明这是一个可执行程序。

一个生成脚本的会话过程　下面的范例中，用户将在编辑器中创建一个脚本。保存文件之后，用户打开脚本的执行权限，然后执行它。如果程序中有任何错误，shell 将立刻做出反应。

范例 8-2

(脚本)

```
1   #!/bin/sh
2   # Scriptname: greetings
    # Written by:  Barbara Born
    # This is the first Bourne shell program of the day.
3   echo "Hello $LOGNAME, it's nice talking to you."
4   echo "Your present working directory is `pwd`."
    echo "You are working on a machine called `uname -n`."
    echo "Here is a list of your files."
5   ls      # List files in the present working directory
6   echo "Bye for now $LOGNAME. The time is `date +%T`!"
```

(命令行)

```
7   $ chmod +x greetings
    $ greetings
3   Hello barbara, it's nice talking to you.
4   Your present working directory is /home/lion/barbara/prog
    You are working on a machine called lion.
    Here is a list of your files.
5   Afile        cplus   letter   prac
    Answerbook   cprog   library  prac1
    bourne       joke    notes    perl5
6   Bye for now barbara. The time is 18:05:07!
```

说明

1. 脚本的第一行，#!/bin/sh，告诉内核要用哪个解释器来执行这个程序，本例中用的是 sh(Bourne shell)解释器。

2. 注释是跟在#号后的不可执行的行。它们可以自成一行，也可以附加在命令后与其共处一行。

3. shell 执行变量替换之后，echo 命令把这一行显示在屏幕上。

4. shell 执行命令替换之后，echo 命令把这一行显示在屏幕上。

5. 执行 ls 命令。注释将被 shell 忽略。

6. echo 命令显示双引号括着的字符串。双引号内的变量和命令替换(反引号)也会被展

开。本例中也可以不使用双引号。

7. greetings 脚本为用户、组、和其他成员指定可执行权限。并且 greetings 脚本是从命令行运行的。

8.2 读取用户输入

read 命令是一个内置命令，用于从终端或文件读取输入(参见表 8-1)。read 命令将读取一个输入行，直至遇到换行符。行尾的换行符在读入时将被转换成一个空字符。你也可以用 read 命令来中断程序的运行，直至用户输入一个回车。想要知道如何最有效地使用 read 命令从文件读取输入行，请参见 8.6 节，“循环命令”。

表 8-1 read 命令

格 式	含 义
read answer	从标准输入读取一行并赋值给变量 answer
read first last	从标准输入读取一行，直至遇到第一个空白符或换行符。把用户键入的第一个词存到变量 first 中，把该行的剩余部分保存到变量 last 中

范例 8-3

(脚本)

```
    # !/bin/sh
    # Scriptname: nosy
    echo "Are you happy? \c"
1   read answer
    echo "$answer is the right response."
    echo "What is your full name? \c"
2   read first middle last
    echo "Hello  $first"
-------------------------------------------------------------
```

(输出)

```
    Are you happy? Yes
1   Yes is the right response.
2   What is your full name? Jon Jake Jones
    Hello Jon
```

说明

1. read 命令接收一行用户输入，将其赋值给变量 answer。

2. read 命令从用户处接收输入，将输入的第一个词赋给变量 first，将第二个词赋给变量 middle，然后将直到行尾的所有剩余单词都赋给变量 last。

范例 8-4

(脚本)

```
    #!/bin/sh
    # Scriptname: printer_check
```




```
    # Script to clear a hung up printer for SVR4
1   if [ $LOGNAME != root ]
    then
       echo "Must have root privileges to run this program"
       exit 1
    fi
2   cat << EOF
    Warning: All jobs in the printer queue will be removed.
    Please turn off the printer now. Press Enter when you
    are ready to continue. Otherwise press Ctrl-C.
    EOF
3   read ANYTHING      # Wait until the user turns off the printer
    echo
4   /etc/init.d/lp stop      # Stop the printer
5   rm -f /var/spool/lp/SCHEDLOCK /var/spool/lp/temp*
    echo
    echo "Please turn the printer on now."
6   echo "Press Enter to continue"
7   read ANYTHING           # Stall until the user turns the printer back on
    echo                    # A blank line is printed
8   /etc/init.d/lp start  # Start the printer
```

说明

1. 检查用户是否为 root。如不是，则发送错误信息并退出。

2. 生成 here 文档。在屏幕上显示警告信息。

3. read 命令等待用户输入。当用户按下回车键时，变量 ANYTHING 接收用户之前键入的内容。这个变量没什么实际用处。此处的 read 用来等待用户关闭打印机，然后回来按下回车键。

4. lp 命令终止打印机守护进程。

5. 重启调度程序之前，必须先删除文件 SCHEDLOCK，以及目录/var/spool/lp 下的临时文件。

6. 请用户在准备好之后按下回车键。

7. 用户键入的任何内容都被读入变量 ANYTHING，用户按下回车键之后，程序恢复运行。

8. lp 程序启动打印守护进程。

8.3 算术运算

Bourne shell 没有内置算术运算。如果要执行简单的整数算术运算，Bourne shell 脚本中最常用的是 UNIX 的 expr 命令。如果要执行浮点算术运算，则可使用 awk 或 bc 程序。因为没有内置算术运算，进行多次循环时，shell 的性能会降低。循环机制中每次增减计数器时，shell 都必须派生一个进程来处理算术运算。

8.3.1 整数运算与 expr 命令

expr 命令是一个处理表达式的程序。用于计算算术表达式时，expr 能执行简单的整数运算(参见表 8-2)。expr 的每个参数之间必须用空格分隔。expr 支持+、-、*、/和%这几个运算符，还能应用关于结合性和优先级的标准编程规则。

表 8-2 expr 命令的算术运算符

运 算 符	功 能
*	乘法
/	除法
%	取模
+	加法
-	减法

范例 8-5

```
1  $ expr 1 + 4
   5
2  $ expr 1+4
   1+4
3  $ expr 5 + 9  /  3
   8
4  $ expr 5 * 4
   expr: syntax error
5  $ expr 5 \* 4  -  2
   18
6  $ expr 11 % 3
   2
7  $ num=1
   $ num=`expr $num + 1`
   $ echo $num
   2
```

说明

1. expr 命令用于计算该表达式。将两个数相加。
2. 由于操作数之间未加空格，该表达式被当成一个字符串。
3. 加法和除法相结合。先执行除法运算，然后再做加法。
4. 星号(*)被 shell 当作通配符展开，导致 expr 命令失败。
5. 用反斜杠转义星号，以免其被 shell 解释。expr 命令执行运算。
6. 模运算符(%)执行除法，然后返回余数。
7. 变量 num 被赋值为 1。expr 命令将变量 num 的值加 1，然后赋回给 num。num 的值被回显在屏幕上。

8.3.2 浮点运算

执行更复杂的运算时，bc、awk 和 nawk 这些工具很有用。



范例 8-6

(命令行)

```
1   $ n=`echo "scale=3; 13 / 2" | bc`
    $ echo $n
    6.500
2   $ n=`bc << EOF
    scale=3
    13/2
    EOF`
    $ echo $n
    6.500
3   $ product=`nawk -v x=2.45 -v y=3.123 'BEGIN{printf "%.2f\n",x*y}'`
    $ echo $product
    7.65
```

说明

1. echo 命令的输出被管道发给程序 bc。scale 被设为 3，scale 是打印结果时小数点后的有效位数。执行的计算是 13 除以 2。整个管道命令被括在双引号中。shell 将执行命令替换并将结果赋给变量 n。

2. 这里用 here 文档加反斜杠来实现与第 1 行相同的功能。命令的结果被赋值给变量 n，然后由 echo 命令显示。

3. nawk 程序从命令行传来的参数列表中获取操作数的值：x=2.45 y=3.123(-v 选项只适用于 nawk，不适用于 awk)。nawk 将两个数相乘，然后用 printf 函数设置好格式并打印结果，精确到小数点后两位。nawk 命令的输出被赋给变量 product。

8.4 位置参量和命令行参数

用户可以通过命令行向脚本传递信息。脚本名后(用空白符分隔的)每个词都称为参数。

我们可以在脚本中使用位置参量来引用命令行参数，例如，$1 代表第 1 个参数，$2 代表第 2 个参数，$3 代表第 3 个参数，以此类推。变量$#可以用来测试参量的个数，而$*则可以显示所有的参量。我们可以用 set 命令来设置或重置位置参量。如果使用了 set 命令，之前设置的所有位置参量都会被清空。请参见表 8-3。

表 8-3 Bourne shell 的位置参量

位置参量	指代对象
$0	脚本名
$#	位置参量的个数
$*	所有的位置参量
$@	未加双引号时，与$*的含义相同
"$*"	扩展为单个变量(例如："$1 $2 $3")。
"$@"	扩展为多个单独的变量(例如："$1", "$2", "$3")
$1 … $9	最多可引用 9 个位置参量

范例 8-7

(脚本)

```
    #!/bin/sh
    # Scriptname: greetings
    echo "This script is called $0."
1   echo "$0  $1 and $2"
    echo "The number of positional parameters is $#"
    -----------------------------------------------------------
```

(命令行)

```
    $ chmod +x greetings
2   $ greetings
    This script is called  greetings.
    greetings  and
    The number of positional parameters is
3   $ greetings Tommy
    This script is called greetings.
    greetings Tommy and
    The number of positional parameters is 1
4   $ greetings Tommy  Kimberly
    This script is called greetings.
    greetings Tommy and Kimberly
    The number of positional parameters is 2
```

说明

1. 在脚本 greetings 中，位置参量$0 指代脚本名，$1 指代第 1 个命令行参数，$2 则指代第 2 个命令行参数。

2. 执行脚本 greetings，不带任何参数。输出结果说明：脚本名为 greetings(脚本中的$0)，$1 和$2 没有被赋值，所以它们的值为空，没有什么可打印。

3. 这次传递了一个参数：Tommy。Tommy 被赋给第 1 个位置参量。

4. 传入两个参数：Tommy 和 Kimberly。Tommy 被赋给$1，Kimberly 则被赋给$2。

8.4.1 set 命令与位置参量

带参数的 set 命令将重置位置参量[1]。位置参量一旦被重置，原来的参量列表就会丢失。要想清除所有的位置参量，可使用命令 set --. $0 始终指代脚本名。

范例 8-8

(脚本)

```
    #!/bin/sh
    # Scriptname: args
    # Script to test command-line arguments
1   echo The name of this script is $0.
2   echo The arguments are $*.
3   echo The first argument is $1.
```

① 注意，不带参数时，set 命令显示为该 shell 设置的所有变量，局部变量和输出变量都包括在内。带选项时，set 命令可以打开或关闭 shell 的控制选项，例如-x 和-v。



```
4   echo The second argument is $2.
5   echo The number of arguments is $#.
6   oldargs=$*             # Save parameters passed in from the command line
7   set Jake Nicky Scott   # Reset the positional parameters
8   echo All the positional parameters are $*.
9   echo The number of postional parameters is $#.
10  echo "Good-bye for now, $1 "
11  set `date`             # Reset the positional parameters
12  echo The date is $2 $3, $6.
13  echo "The value of \$oldargs is $oldargs."
14  set $oldargs
15  echo $1 $2 $3
```

(命令行和输出)

```
    $ args a b c d
1   The name of this script is args.
2   The arguments are a b c d.
3   The first argument is a.
4   The second argument is b.
5   The number of arguments is 4.
8   All the positional parameters are Jake Nicky Scott.
9   The number of positional parameters is 3.
10  Good-bye for now, Jake
12  The date is Mar 25, 2004.
13  The value of $oldargs is a b c d.
15  a b c
```

说明

1. 脚本的名称保存在变量$0 中。

2. $*代表所有的位置参量。

3. $1 代表第 1 个位置参量(命令行参数)。

4. $2 代表第 2 个位置参量。

5. $#是位置参量(命令行参数)的总个数。

6. 把所有的位置参量都保存在变量 oldargs 中。

7. set 命令重置位置参量，清空原来的位置参量列表。现在，$1 是 Jake，$2 是 Nicky，$3 则是 Scott。

8. $*代表所有的位置参量，即 Jake、Nicky 和 Scott。

9. $#代表位置参量的个数，即 3。

10. $1 是 Jake。

11. 执行命令替换之后，即执行 date 命令后，位置参量被重置为 date 命令的输出。

12. 显示$2、$3 和$6 的新值。

13. 打印保存在 oldargs 中的值。

14. set 命令根据 oldargs 中保存的值创建位置参量。

15. 显示前 3 个位置参量。

范例8-9

(脚本)

```
    #!/bin/sh
    # Scriptname: checker
    # Script to demonstrate the use of special variable modifiers and
    arguments
1   name=${1:?"requires an argument" }
    echo Hello $name
```

(命令行)

```
2   $ checker
    ./checker: 1: requires an argument
3   $ checker Sue
    Hello Sue
```

说明

1. 特殊变量修饰符:?将检查$1 是否有值。如果$1 无值，则打印指定信息并退出。

2. 不带参数执行该程序。$1 没有被赋值，程序显示报错信息。

3. 给 checker 程序一个命令行参数，即 Sue。在脚本中，$1 被赋值为 Sue。程序继续运行。

8.4.2 $*和$@有何区别

$*和$@仅在被双引号括起来时有区别。$*被括在双引号中时，位置参量列表就变成单个字符串。而$@被括在双引号中时，每个位置参量都被加上引号，也就是说，每个词都被视作一个单独的字符串。

范例8-10

```
1   $ set 'apple pie' pears peaches
2   $ for i in $*
    > do
    > echo $i
    > done
    apple
    pie
    pears
    peaches
3   $ set 'apple pie' pears peaches
4   $ for i in "$*"
    > do
    > echo $i
    > done
    apple pie pears peaches
5   $ set 'apple pie' pears peaches
6   $ for i in $@
    > do
    > echo $i
    > done
    apple
    pie
    pears
```



```
    peaches
7   $ set 'apple pie' pears peaches
8   $ for i in "$@"        # At last!!
    > do
    > echo $i
    > done
    apple pie
    pears
    peaches
```

说明

1. 设置位置参量。

2. $*展开后，apple pie 两端的引号被去掉，apple 和 pie 变成了两个独立的词。for 循环将每个词依次赋给变量 i，然后打印 i 的值。每执行一次循环，都将左端的词移走，将下一个词赋给变量 i。

3. 设置位置参量。

4. 给$*加上引号后，整个参量列表就变成了一个字符串，即"apple pie pears peaches"。整个列表只作为一个词被赋给 i。只执行一次循环。

5. 设置位置参量。

6. 没有加引号时，$@和$*的表现一致(参见第 2 条说明)。

7. 设置位置参量。

8. 给$@加上双引号后，每个位置参量都被视为一个加引号的字符串。列表将变成"apple pie"、"peares"、"peaches"，从而得到理想的结果。

8.5 条件结构和流控制

条件结构使你能够根据是否满足某个特定条件来选择执行相应操作。if 命令是最简单的决策形式。if/else 命令提供双路决策，而 if/elif/else 命令则提供多路决策。

Bourne shell 要求 if 后面必须跟一条命令，可以是系统命令，也可以是内置命令。命令的退出状态被用于条件的求值。

计算表达式时要使用 Bourne shell 的内置命令 test。这个命令也已链接到方括号这个符号上。因此，既可以用 test 命令，也可以将表达式括在方括号中来对表达式求值。test 命令不会展开 shell 的元字符(通配符)。test 测试命令的结果之后，返回状态 0 表示成功，返回非 0 状态则表示失败。参见表 8-4。

表 8-4　字符串、整数和文件测试

测试运算符	测 试 内 容
字符串测试	
string1 = string2	string1 等于 string2(=两侧必须有空格)
string1 != string2	string1 不等于 string2(!=两侧必须有空格)
string	string 不为空

(续表)

测试运算符	测 试 内 容
-z string	string 的长度为 0
-n string	string 的长度不为 0
例子：	test –n $word 或 [-n $word]
	test tom = sue 或 [tom = sue]
整数测试	
int1 –eq int2	int1 等于 int2
int1 –ne int2	int1 不等于 int2
int1 –gt int2	int1 大于 int2
int1 –ge int2	int1 大于或等于 int2
int1 –lt int2	int1 小于 int2
int1 –le int2	int1 小于或等于 int2
逻辑测试	
expr1 -a expr2	逻辑与
expr1 -o expr2	逻辑或
! expr	逻辑非
文件测试	
-b filename	该文件是块特殊文件
-c filename	该文件是字符特殊文件
-d filename	该目录存在
-f filename	该普通文件存在且不是目录
-g filename	设置了 set-group-ID 位
-k filename	sticky 位被设置
-p filename	该文件是命名管道
-r filename	该文件可读
-s filename	文件大小不为 0
-u filename	设置了 set-user-ID 位
-w filename	该文件可写
-x filename	该文件可执行

8.5.1 测试退出状态：test 命令

下面的范例说明如何测试退出状态。

test 命令用于计算条件表达式，返回真或假。test 命令返回的退出状态为 0 代表真，非 0 代表假。test 命令或方括号都可用来执行测试操作(参见表 8-4)。

范例 8-11
(命令行)




```
1  $ name=Tom
2  $ grep "$name"  /etc/passwd
   Tom:8ZKX2F:5102:40:Tom Savage:/home/tom:/bin/ksh
3  $ echo $?
   0           Success!
4  $ test $name != Tom
5  $ echo $?
   1           Failure
6  $ [ $name = Tom ]        # Brackets replace the test command
7  $ echo $?
   0           Success
8  $ [ $name =  [Tt]?m ]     # Wildcards are not evaluated by the test command
9  $ echo  $?
   1           Failure
```

说明

1. 变量 name 被赋值为字符串 Tom。

2. grep 命令在文件 passwd 中查找字符串 Tom。

3. 变量? 包含 shell 执行的上一条命令的退出状态，本例中是 grep 的退出状态。如果 grep 成功地找到了字符串 Tom，它返回退出状态 0。因此，这条 grep 命令执行成功。

4. test 命令可用于计算字符串和数字，也可用来执行文件测试。和所有命令一样，test 也会返回一个退出状态：退出状态为 0，则表达式为真，退出状态为 1，则表达式为假。表达式的等号两侧必须有空格。这条命令测试 name 的值是否等于 Tom。

5. 测试失败，test 返回的退出状态为 1。

6. 方括号是 test 命令的另一种表示方式。第一个方括号后面必须跟空格。这一行测试 $name 的值是否为字符串 Tom。

7. test 的返回值是 0。因为$name 等于 Tom，所以 test 返回成功。

8. test 命令不允许展开通配符。问号被当作一个普通字符，因此测试失败，Tom 与 [Tt]?m 不相等。

9. 退出状态为 1，表示第 8 行的 test 命令返回失败。

8.5.2 if 命令

if 命令是形式最简单的条件结构。跟在 if 结构后面的命令或 UNIX 工具被执行，并返回其退出状态。程序的退出状态通常由编程人员决定。退出状态为 0，表示命令执行成功，shell 将执行关键字 then 后面的语句。在 C shell 中，跟在 if 命令后的表达式是布尔型的表达式，与 C 语言一样。在 Bourne shell 和 Korne shell 中，跟在 if 后面的则是一条或一组命令。如果该命令的退出状态为 0，shell 就执行从 then 到 fi 之间的语句块。关键字 fi 用来结束 if 结构。如果退出状态非 0，说明命令由于某种原因运行失败，则 shell 忽略关键字 then 后的语句，控制跳到紧跟在 fi 语句后面的那条语句。重要的是可以因此知道被测试命令的退出状态。例如，grep 的退出状态能够准确地告诉您 grep 是否在文件中找到了它所查找的模式：如果查找成功，grep 返回退出状态 0，不成功则返回 1。sed 和 awk 程序也查找模式，但是不论是否找到模式，它们都报告一个成功的返回状态。sed 和 awk 判断成功的标准是

语法是否正确，而不是功能[②]。

格式

```
if 命令
then
  命令
  命令
fi

if test 表达式
then
  命令
fi
  或
if [ 表达式 ]
then
  命令
fi
```

范例 8-12

```
1   if ypmatch "$name" passwd > /dev/null 2>&1
2   then
        echo Found $name!
3   fi
```

说明

1. ypmatch 命令是一条 NIS 命令(NIS 是 Sun 公司的网络信息服务，Network Information Services)，该命令在位于服务器上的 NIS 口令数据库中查找命令的参数 name。标准输出和标准错误输出都被重定向到 UNIX 的位桶/dev/null 中。如果您的系统不支持 ypmatch 命令，可以试试下面这条命令：

```
if grep "$name" /etc/passwd > /dev/null 2>&1
```

2. 如果 ypmatch 命令的退出状态为 0，程序将转向 then 语句，执行其后的语句，直到到达 fi 为止。

3. fi 标志着 then 语句后的命令序列结束。

范例 8-13

```
1   echo "Are you okay (y/n) ?"
    read answer
2   if [ "$answer" = Y -o "$answer" = y  ]
    then
        echo "Glad to hear it."
3   fi
```

说明

1. 程序向用户提出问题并要求回答。read 命令等候用户的响应。

② 与 C shell 不同的是，Bourne shell 不支持不带 then 的 if 语句，再简单的 if 语句后面也要包含关键字 then。



2. 方括号代表 test 命令，用于测试表达式。如果表达式为真，该命令返回 0，如果表达式为假，则返回 1。如果求得变量 answer 的值为 Y 或 y，则执行语句 then 后面的命令(测试表达式时，test 命令不允许使用通配符，并且要求方括号和等号两侧都必须有空格)。参见前面的表 8-4。

变量$answer 需被括在双引号中以作为单个字符串。因为如果等号左边出现不止一个单词时会导致 test 命令失败。例如，若用户输入 yes，you betcha，则 answer 变量会等价于 3 个单词。结果会使得 test 测试失败。因此，$answer 必须加双引号。

3. fi 用来终止 then 语句后的命令序列。

8.5.3 exit 命令和?变量

exit 命令用于终止脚本并返回命令行。您可能希望脚本在某些情况发生时退出。传给 exit 命令的参数是一个从 0~255 的整数。如果程序以 0 为参数退出，则表明程序执行成功，参数非 0 则表示程序执行失败。传给 exit 命令的参数被保存在 shell 的变量？中。

范例 8-14

(脚本)

```
    # Name: bigfiles
    # Purpose: Use the find command to find any files in the root
    # partition that have not been modified within the past n (any
    # number within 30 days) days and are larger than 20 blocks
    # (512-byte blocks)
1   if  [ $# -ne 2 ]
    then
       echo  "Usage: $0 mdays size " 1>&2
       exit 1
2   fi
3   if  [ $1 -lt 0 -o $1 -gt 30 ]
    then
       echo "mdays is out of range"
       exit 2
4   fi
5   if [ $2 -le 20 ]
    then
       echo "size is out of range"
       exit 3
6   fi
7   find / -xdev -mtime $1 -size +$2 -print
```

(命令行)

```
$ bigfiles
Usage: bigfiles mdays size
$ echo $?
1
$ bigfiles 400 80
mdays is out of range
$ echo $?
2
```

```
$ bigfiles 25 2
size is out of range
$ echo $?
3
$ bigfiles 2 25
(find 的输出)
```

说明

1. 这条语句的含义是：如果参数的个数不等于 2，则打印报错信息并将其发给标准错误输出，然后返回状态值 1 并退出脚本。

2. fi 标志了 then 后面的语句块的结束。

3. 这条语句的含义是：如果从命令行传入的第 1 个位置参量的值小于 0 或大于 30，就打印报错信息，并以状态 2 退出。有关数值运算符的内容，请参见前面的表 8-14。

4. fi 结束 if 语句块。

5. 这条语句的含义是：如果从命令行传入的第 2 个位置参量的值大于或等于 20(一个 512 字节的块)，则打印报错信息，并以状态 3 退出。

6. fi 结束 if 语句块。

7. find 命令从根目录开始搜索。选项-xdev 阻止 find 搜索其他分区；选项-mtime 带一个数字参数，这个参数代表自文件最后一次被修改以来的天数；选项-size 也带一个数字参数，它表示以 512 字节的块为单位计算而得的文件大小。

8.5.4 检查空值

检查变量的值是否为空时，必须用双引号把空值括起来，否则 test 命令就会失败。

范例 8-15

```
(脚本)
1   if [ "$name" = "" ]    # Alternative to  [ ! "$name" ]  or  [ -z "$name" ]
    then
       echo The name variable is null
    fi
(通过系统的 showmount 程序，显示了所有已远程装载的系统)
    remotes=`/usr/sbin/showmount`
2   if [ "X${remotes}" != "X" ]
    then
       /usr/sbin/wall ${remotes}
                    ...
3   fi
```

说明

1. 如果变量 name 的值为空，则测试结果为真。用双引号代表空值。

2. showmount 命令列出了远程装载到指定主机上的所有客户。这条命令可能会列出一个或多个客户，也可能没有输出。于是变量 remotes 可能被赋值，也可能为空。进行测试时，变量 remotes 前面加了一个字母 X。如果 remotes 的值为空，说明没有远程登录过来的客户，于是 X 等于 X，使得程序从标注了 3 的行开始执行。如果变量 remotes 有值，比如



主机名 pluto，则表达式变成 if Xpluto != X，于是执行 wall 命令(向远程主机上的所有用户发送消息)。在表达式中使用 X 的目的是：确保即使在 remote 值为空的情况下，表达式中运算符!=的两边也都有一个占位符。

3. fi 结束 if 语句块。

8.5.5 if/else 命令

if/else 命令提供双路决策操作。如果 if 后面的命令失败了，就执行 else 后面的命令。

格式

```
if 命令
then
 命令(命令组)
else
 命令(命令组)
fi
```

范例 8-16

(脚本)

```
   #!/bin/sh
1  if ypmatch "$name" passwd > /dev/null 2>&1③
2  then
      echo Found $name!
3  else
4     echo  "Can't find $name."
      exit 1
5  fi
```

说明

1. ypmatch 命令在 NIS passwd 数据库中查找它的参数 name。因为用户不需要看输出结果，所以标准输出和标准错误输出都被重定向到 UNIX 的位桶/dev/null 中。

2. 如果 ypmatch 命令的退出状态为 0，程序的控制转向 then 语句，并执行其后的语句直到遇到 else 为止。

3. 如果 ypmatch 命令没有在 passwd 数据库中找到$name，就执行 else 语句后的命令。也就是说，只有在 ypmatch 命令的退出状态不为 0 时，才执行 else 块。

4. 如果未在 passwd 数据库中找到$name 的值，就执行 echo 语句，之后程序以状态 1 退出，说明查找失败。

5. fi 结束 if 语句块。

范例 8-17

(脚本)

```
#!/bin/sh
# Scriptname: idcheck
# purpose: check user ID to see if user is root.
```

---

③ 如果使用的是 NIS+，则命令为：if nismatch "$name" passwd.org_dir。

```
    # Only root has a uid of 0.
    # Format for id output:uid=9496(ellie) gid=40 groups=40
    # root's uid=0
1   id=`id | nawk -F'[=(]' '{print $2}'`     # Get user ID
    echo your user id is: $id
2   if [ $id -eq 0 ]
    then
3       echo "you are superuser."
4   else
        echo "you are not superuser."
5   fi
    (命令行)
6   $ idcheck
    your user id is: 9496
    you are not superuser.
7   $ su
    Password:
8   # idcheck
    your user id is: 0
    you are superuser
```

说明

1. id 命令的输出通过管道发送给 nawk 命令。nawk 使用等号或左圆括号作为字段分隔符，从 id 命令的输出结果中提取出用户的 ID，并把结果赋给变量 id。
2. 如果变量 id 的值等于 0，则执行第 3 行。
4. 如果 id 不等于 0，则执行 else 后的语句。
5. fi 标志 if 命令的结束。
6. 由 UID 为 9496 的当前用户执行 idcheck 脚本。
7. su 命令将用户切换为 root。
8. 命令提示符#表示超级用户(root)成为当前的新用户。root 的 UID 是 0。

8.5.6 if/elif/else 命令

if/elif/else 命令提供多路决策操作。如果 if 后的命令失败了，则测试 elif 后的命令。如果这条命令成功，就执行它的 then 语句后面的命令。如果 elif 后面的命令也失败了，就检查下一条 elif 命令。如果所有的 elif 命令都不成功，则执行 else 命令。else 操作块被称为默认操作。

格式

```
if 命令
then
   命令(命令组)
elif 命令
then
   命令(命令组)
elif 命令
then
```




```
   命令(命令组)
else
   命令(命令组)
fi
```

范例 8-18

(脚本)

```
    #!/bin/sh
    # Scriptname: tellme
1   echo -n "How old are you? "
    read age
2   if [  $age -lt  0 -o $age -gt 120 ]
    then
       echo  "Welcome to our planet! "
       exit 1
    fi
3   if [ $age -ge 0 -a  $age -lt 13 ]
    then
       echo "A child is a garden of verses"
    elif [ $age -ge 13 -a $age -lt  20 ]
    then
       echo "Rebel without a cause"
    elif [ $age  -ge 20 -a  $age -lt  30 ]
    then
       echo "You got the world by the tail!!"
    elif [ $age -ge  30 -a  $age -lt 40 ]
    then
       echo "Thirty something..."
4   else
       echo "Sorry I asked"
5   fi
```

(输出)

```
$ tellme
How old are you? 200
Welcome to our planet!
$ tellme
How old are you? 13
Rebel without a cause
$ tellme
How old are you? 55
Sorry I asked
```

说明

1. 要求用户输入，将用户的输入赋给变量 age。

2. 执行方括号内的数值测试。如果 age 小于 0 或大于 120，就执行 echo 命令，然后程序以状态 1 终止。屏幕上将出现交互式 shell 的提示符。

3. 执行方括号内的数值测试。如果 age 大于 0 并且小于 13，test 命令就返回退出状态 0，即真，并且执行 then 后面的语句。否则，程序控制转到 elif。如果 elif 的测试结果为假，就测试下一个 elif。

4. else 结构是默认操作。如果之前的语句都不为真，则执行 else 语句块中的命令。
5. fi 结束从第 3 行开始的 if 语句。

8.5.7 文件测试

写脚本的时候，常常需要使用某些特定文件，并且要求这些文件有特定的权限、属于某个类型或具有其他一些属性(参见表 8-4)。下面，您将会发现文件测试是编写可靠脚本的一个必要部分。

当 if 语句出现嵌套时，fi 语句总是对应离它最近的 if 语句。以逐层缩进方式编排的 if 嵌套语句可以使 if 语句和 fi 语句的对应关系看起来更清楚。

范例 8-19

(脚本)
```
    #!/bin/sh
    file=./testing
1   if [ -d $file ]
    then
      echo "Sfile is a directory"
2   elif [ -f $file ]
    then
3     if [ -r $file -a -w $file -a -x $file ]
      then  # nested if command
          echo "You have read, write, and execute permission on $file,
4     fi
5   else
      echo "$file is neither a file nor a directory,"
6   fi
```

说明

1. 如果文件 testing 是一个目录，就输出 "testing is a directory"。
2. 如果文件 testing 不是目录，或者文件是普通文件，则执行 then 语句块。
3. 如果文件 testing 可读、可写而且可执行，则执行 them 语句块。
4. fi 结束最内层的 if 命令。
5. 如果第 1 行和第 2 行都不为真，则执行 else 中的命令。
6. 这个 fi 对应第一个 if。

8.5.8 null 命令

冒号代表的 null 命令是 shell 的一个内置命令，它不做任何事情，只返回退出状态 0。null 命令有时被放在 if 命令后面作为一个占位符，这时 if 命令不执行实际操作，只需要在 then 后面放置一条命令，否则，程序会生成报错信息，因为 then 语句后面必须添加内容。null 命令常常被用作循环命令的参数，其作用是让循环无限地进行下去。

范例 8-20

(脚本)
```
1   name=Tom
2   if grep "$name" databasefile > /dev/null 2>&1
```



```
      then
3         :
4     else
          echo  "$1 not found in databasefile"
          exit 1
      fi
```

说明

1. 变量 name 被赋值为字符串 Tom。

2. if 命令测试 grep 命令的退出状态。如果在 databasefile 中找到了字符串 Tom，就执行 null 命令(即冒号)，但该命令不做任何操作。

3. 冒号是 null 命令。除了返回退出状态 0 外，这条命令不做任何操作。

4. 我们真正要做的是：如果没找到 Tom，就输出一条报错信息并退出。若 grep 命令运行失败，则执行 else 后面的命令。

范例 8-21

(命令行)

```
1   $ DATAFILE=
2   $ : ${DATAFILE:=$HOME/db/datafile}
    $ echo $DATAFILE
    /home/jody/ellie/db/datafile
3   $ : ${DATAFILE:=$HOME/junk}
    $ echo $DATAFILE
    /home/jody/ellie/db/datafile
```

说明

1. 变量 DATAFILE 被赋值为空。

2. 冒号命令不执行任何实际操作。修饰符(:=)返回一个可以赋给变量或用于测试的值。此例子中，表达式将作为变量赋给这条冒号命令。shell 会执行变量替换，即若此时 DATAFILE 尚未赋值，则将路径名赋给它。这样，变量 DATAFILE 就始终都会有值。

3. 由于变量 DATAFILE 已经被赋了值，所以 shell 不会再用修饰符:=右边提供的默认值来重置它。

范例 8-22

(脚本)

```
    #!/bin/sh
1   # Name:wholenum
    # Purpose:The expr command tests that the user enters an integer
    echo "Enter a number."
    read number
2   if expr "$number" + 0 > /dev/null 2>&1
    then
3      :
    else
4      echo "You did not enter an integer value." 1782
       exit 1
5    fi
```

说明

1. 要求用户输入一个整数。将该整数赋给变量 number。

2. expr 命令对表达式求值。如果加法能顺利执行，则说明用户输入的确实是一个整数，expr 就返回成功的退出状态。且所有输出都被重定向到位桶/dev/null 中。

3. 若 expr 命名执行成功，则返回退出状态 0，而冒号命令不做任何操作。

4. 若 expr 命名执行失败，则返回非 0 的退出状态，先由 echo 命令显示消息到指定的标准错误输出(1782)，然后程序退出。

5. fi 结束 if 块。

8.5.9 case 命令

case 命令是一个多路分支命令，可用来代替 if/elif 命令。case 变量的值与值 1，值 2 等值逐一比较，直至找到与之匹配的值。如果某个值与 case 变量匹配，程序就执行该值后面的命令，直至遇到双分号，然后跳到词 esac(case 倒过来拼写)后面继续往下执行。

如果没有找到与 case 变量匹配的值，程序就执行默认值*)后面的命令，直至遇到;;或 esac。值*)的功能与 if/else 条件中 else 语句的相同。case 的值里可以用 shell 通配符，还可以用竖杠(管道符)对两个值取或。

格式

```
case 变量 in
值 1)
     命令(命令组)
    ;;
值 2)
    命令(命令组)
    ;;
*)
 命令(命令组)
   ;;
esac
```

范例 8-23

(脚本)

```
   #!/bin/sh
   # Scriptname: colors
1  echo -n "Which color do you like?"
   read color
2  case "$color" in
3  [Bb]l??)
4     echo I feel $color
      echo The sky is $color
5     ;;
6  [Gg]ree*)
      echo  $color is for trees
      echo  $color is for seasick;;
7  red | orange)            # The vertical bar means "OR"
```



```
        echo $color is very warm!;;
8   *)
        echo No such color as $color;;
9   esac
10  echo "Out of case"
```

说明

1. 要求用户输入。将输入赋值给变量 color。

2. case 命令对表达式$color 求值。

3. 如果变量 color 的值以 B 或 b 开头，后面跟字母 l 和两个任意字符，则 case 表达式匹配第一个值。case 的值以单个圆括号终止。这些通配符是用于文件名扩展的 shell 元字符。

4. 如果第 3 行的值与 case 表达式相匹配，则执行这两条语句。

5. 命令块的最后一行必须有双分号。程序执行到双分号时，就会将控制跳转到第 10 行。注意，把双分号单独写在一行可以方便对脚本的调试。

6. 如果 case 表达式匹配以 G 或 g 开头，后跟字母 ree，并以零个或多个任意字符结尾的值，则执行 echo 命令。双分号用于结束该语句块，且控制跳转到第 10 行。

7. 竖杠用作条件运算符或。如果 case 表达式匹配 red 或 orange，则执行 echo 命令。

8. 这是默认值。如果以上所有值都不能匹配 case 表达式，就执行*)后面的命令。

9. esac 语句结束 case 命令。

10. 匹配完 case 任一值后，程序将跳转到这里继续执行。

8.5.10 用 here 文档和 case 命令生成菜单

here 文档常常与 case 命令搭配使用。我们可以用 here 文档生成一个显示在屏幕上的选项菜单，要求用户选择一个菜单项，然后用 case 命令对照选项集测试用户的输入，以执行相应的命令。

范例 8-24

```
( .profile 文件)
    echo "Select a terminal type: "
1   cat << ENDIT
        1) vt 120
        2) wyse50
        3) sun
2   ENDIT
3   read choice
4   case "$choice" in
5   1)   TERM=vt120
         export TERM
         ;;
    2)  TERM=wyse50
        export TERM
        ;;
6   3)   TERM=sun
       export TERM
        ;;
```

```
7  esac
8  echo "TERM is $TERM."
(输出)
$ . .profile
Select a terminal type:
1) vt120
2) wyse50
3) sun
3          <-- User input
TERM is sun.
```

说明

1. 如果把这段脚本放在.profile 中，当您登录成功后，就有机会选择正确的终端类型。here 文档用来显示选项菜单。

2. 用户自定义的终止符 ENDIT 标志 here 文档的结束。

3. read 命令把用户的输入保存到变量 choice 中。

4. case 命令求出变量 choice 的值，然后将其与下面右圆括号前面的值(1、2 或 3)逐一进行比较。

5. 测试的第一个值是 1。如果二者匹配，就将终端类型设置为 vt120。然后输出变量 TERM，以使子 shell 能够继承它。

6. 不需要默认值。通常是在登录时，在/etc/profile 中设置变量 TERM。如果用户选择 3，则将终端类型设置为 sun。

7. esac 结束 case 命令。

8. case 命令结束后，就执行这一行语句。

8.6 循环命令

循环命令按指定次数执行某条或某组命令，或者一直执行，直到满足某个条件为止。Bourne shell 提供了 3 种类型的循环，即：for 循环、while 循环和 until 循环。

8.6.1 for 命令

for 循环命令在某个对象列表上按指定次数执行命令。譬如，您可能会用 for 循环在某个文件或用户名列表上重复执行相同的命令。for 命令后面跟一个用户自定义的变量、关键字 in 和一列词。执行第一轮循环时，先将词表中的第一个词赋给变量，然后从列表中移开。一旦单词赋值给变量，就会进入循环体，执行关键字 do 和 done 之间的命令。下一次进入循环时，则将第二个词赋给变量，再往后的循环也以此类推。循环体从关键字 do 开始，到关键字 done 结束。当词表中所有的词都被移开后，循环就结束了，程序控制从关键字 done 之后继续执行。

格式

```
for 变量 in 词表
```




```
do
  命令(命令组)
done
```

范例 8-25

```
(脚本)
    #!/bin/sh
    # Scriptname: forloop
1   for pal in Tom Dick Harry Joe
2   do
3       echo "Hi $pal"
4   done
5   echo "Out of loop"
(输出)
$ forloop
Hi Tom
Hi Dick
Hi Harry
Hi Joe
Out of loop
```

说明

1. for 循环将遍历这个姓名列表：Tom、Dick、Harry 和 Joe，遍历完一个就移走一个(往左移，并且将它的值赋给用户自定义变量 pal)。一旦所有的词都被移走，词表变空，循环就结束了，程序从关键字 done 之后接着往下执行。执行第一轮循环时，变量 pal 将被赋值为 Tom；第二轮循环时，pal 将被赋值为 Dick；再下一轮，pal 将被赋值为 Harry；最后一轮，pal 将被赋值为 Joe。

2. 词表后面必须跟关键字 do。如果把 do 和词表放在同一行，就必须用分号结束词表。例如：

```
for pal in Tom Dick Harry Joe; do
```

3. 这是循环体。程序把 Tom 赋给变量 pal 之后，就执行循环体中的命令(即关键字 do 和 done 之间的所有命令)。

4. 关键字 done 结束循环。一旦词表中最后那个词(Joe)被赋给变量并移开，就退出循环。

5. 退出循环后，控制由此重新开始。

范例 8-26

```
(命令行)
1   $ cat mylist
    tom
    patty
    ann
    jake
(脚本)
    #!/bin/sh
    # Scriptname: mailer
2   for person in `cat mylist`
```

```
    do
3       mail $person < letter
        echo  $person was sent a letter.
4   done
5   echo "The letter has been sent."
```

说明

1. 显示文件 mylist 的内容。

2. 执行命令替换，文件 mylist 的内容变成了一个词表。执行第一轮时，tom 被赋给变量 person，然后被移开，由 patty 来代替它的位置，之后各轮循环以此类推。

3. 在循环体里，向每个用户邮寄文件 letter 的一份副本。

4. 关键字 done 标志这轮循环的结束。

5. 给表中所有用户都发送邮件之后，程序退出循环，接着执行这一行。

范例 8-27

（脚本）

```
    #!/bin/sh
    # Scriptname: backup
    # Purpose:
    # Create backup files and store them in a backup directory
1   dir=/home/jody/ellie/backupscripts
2   for file in memo[1-5]
    do
3       if [ -f $file ]
        then
            cp $file $dir/$file.bak
            echo "$file is backed up in $dir"
        fi
    done
```

（输出）

```
memo1 is backed up in /home/jody/ellie/backupscripts
memo2 is backed up in /home/jody/ellie/backupscripts
memo3 is backed up in /home/jody/ellie/backupscripts
memo4 is backed up in /home/jody/ellie/backupscripts
memo5 is backed up in /home/jody/ellie/backupscripts
```

说明

1. 给变量 dir 赋值。

2. 词表由当前工作目录中所有名称以 memo 开头、以 1~5 之间的数字结尾的文件组成。各轮循环将文件名逐个赋给变量 file。

3. 进入循环体后，程序将对文件进行检查，以保证它存在并且是一个真正的文件。如果确实如此，就将它复制到目录/home/jody/ellie/backupscripts 中，给文件名加上后缀.bak。

8.6.2 词表中的$*和$@变量

$*和$@扩展的结果几乎完全一样，唯一的不同是当它们被括在双引号中时，$*的值是一个字符串，而$@的值则是一组独立的词。



范例 8-28

(脚本)

```
    #!/bin/sh
    # Scriptname: greet
1   for name in $*        # Same as for name in $@
2   do
        echo Hi $name
3   done
```

(命令行)

```
$ greet Dee Bert Lizzy Tommy
Hi Dee
Hi Bert
Hi Lizzy
Hi Tommy
```

说明

1. $*和$@展开后是一个包含所有位置参量的列表，这个例子中，它们被展开后的结果就是从命令行传入的参数：Dee、Bert、Lizzy 和 Tommy。列表中的每个名字被依次赋给 for 循环的 name 变量。

2. 执行循环体中的命令，直到列表变空为止。

3. 关键字 done 标志循环体的结束。

范例 8-29

(脚本)

```
    #!/bin/sh
    # Scriptname:permx
1   for file          # Empty wordlist
    do
2       if [ -f $file -a ! -x $file ]
        then
3           chmod +x $file
            echo $file now has execute permission
        fi
    done
```

(命令行)

```
4   $ permx *
    addon now has execute permission
    checkon now has execute permission
    doit now has execute permission
```

说明

1. 如果没有为它提供词列表，for 循环就会遍历所有的位置参量。这行等同于 for file in $* 命令。

2. 文件名将来自命令行。shell 将星号(*)展开为当前工作目录中所有的文件名。如果该文件是一个没有执行权限的文本文件，就执行标为 3 的行。

3. 给每个被处理的文件加上执行权限。

4. shell 把命令行里的星号作为通配符进行求值，将它替换为当前目录下的所有文件。然后，shell 把这些文件作为参数传给脚本 permx。

8.6.3 while 命令

while 对紧跟在它后面的那条命令求值，如果该命令的退出状态为 0，就执行循环体内的命令(即关键字 do 和 done 之间的命令)。执行到关键字 done 后，控制回到循环的顶部，while 命令再次检查那条命令的退出状态。循环将一直继续下去，直到 while 计算的那条命令的退出状态非 0 为止。该命令的退出状态非 0 时，程序将从关键字 done 之后开始执行。

格式

```
while 命令
do
  命令(命令组)
done
```

范例 8-30

(脚本)

```
    #!/bin/sh
    # Scriptname: num
1   num=0                        # Initialize num
2   while [ $num -lt 10 ]        # Test num with test command
    do
        echo -n $num
3       num=`expr $num + 1`      # Increment num
    done
    echo "\nAfter loop exits, continue running here"
```

(输出)

```
0123456789
After loop exits, continue running here
```

说明

1. 初始化步骤，将变量 num 设为 0。

2. while 命令后面跟了一条 test 命令(一对方括号)。如果 num 的值小于 10，就进入循环体。

3. 在循环体中，num 的值被加 1。如果 num 的值始终不变，循环就会无休止地重复下去，直至该进程被终止。

范例 8-31

(脚本)

```
    #!/bin/sh
    # Scriptname: quiz
1   echo "Who was the chief defense lawyer in the OJ case?"
    read answer
2   while [ "$answer" != "Johnny" ]
3   do
4       echo "Wrong try again!"
        read answer
5   done
6   echo You got it!
```

(输出)



```
$ quiz
Who was the chief defense lawyer in the OJ case? Marcia
Wrong try again!
Who was the chief defense lawyer in the OJ case? I give up
Wrong try again!
Who was the chief defense lawyer in the OJ case? Johnny
You got it!
```

说明

1. echo 命令向用户提出问题：Who was the chief defense lawyer in the OJ case(OJ 这个案件的首席辩护律师是谁)? read 命令等待用户输入。用户的输入将被保存在变量 answer 中。

2. 进入 while 循环，test 命令(即方括号内的命名)测试该表达式。如果变量 answer 的值不等于字符串 Johnny，就进入循环体，执行关键字 do 和 done 之间的命令。

3. 关键字 do 是循环体的开始。

4. 要求用户重新输入。

5. 关键字 done 标志循环体的结束。控制返回 while 循环的顶部，再次对表达式进行测试。只要$answer 的值不等于 Johnny，循环就会不间断地重复执行下去。一旦用户输入 Johnny，循环就结束。程序控制将跳转到标注为 6 的那一行。

6. 完成循环体的执行后，控制由此开始。

范例 8-32

(脚本)

```
   #!/bin/sh
   # Scriptname: sayit
   echo Type q to quit.
   go=start
1  while [ -n "$go" ]     # Make sure to double quote the variable
   do
2       echo -n I love you.
3       read word
4       if [ "$word" = q -o "$word" = Q ]
        then
            echo "I'll always love you!
            go=
        fi
   done
```

(输出)

```
$ sayit
Type q to quit.
I love you.    <- When user presses the Enter key, the program continues
I love you.
I love you.
I love you.
I love you.q
I'll always love you!
$
```

说明

1. 执行 while 后面的那条命令，并测试它的退出状态。test 命令的选项-n 用来测试字符串是否非空。由于 go 有初值，所以测试成功，test 返回退出状态 0。如果变量 go 没有加双引号，且值又为空，则 test 命令就会显示：

```
go: test: argument expected
```

2. 进入循环。字符串“I love you.”会回显在屏幕上。

3. read 命令等待用户输入。

4. 测试这个表达式。如果用户输入字母 q 或 Q，就显示字符串“I'll always love you!”，并且将变量 go 设为空。再次进入 while 循环时，由于变量 go 为空，所以测试不成功，循环终止。控制跳转到 done 语句后面的那行。本例中，因为 done 后面没有其他要处理的行，所以脚本将终止。

8.6.4 until 命令

until 命令的用法与 while 命令类似，不过 until 命令只在它后面的命令失败时(即命令返回非 0 的退出状态时)，才执行循环语句。执行到关键字 done 时，控制回到循环顶部，until 命令再次检查其后那条命令的退出状态。循环将继续执行，直到 until 测到那条命令的退出状态变成 0 为止。当该命令的退出状态为 0 时，循环退出，程序将从关键字 done 的下一行开始执行。

格式

```
until 命令
do
  命令(或命令组)
done
```

范例 8-33

(脚本)

```
    #!/bin/sh
1   until who | grep linda
2   do
        sleep 5
3   done
    talk linda@dragonwings
```

说明

1. until 循环测试管道中最后一条命令 grep 的退出状态。who 命令用于列出当前有哪些用户登录到这台机器上，它的输出被管道传给 grep。只有在 who 的输出中找到用户 linda 时，grep 命令才返回退出状态 0(成功)。

2. 如果用户 linda 还没有登录，就进入循环体，程序则挂起 5 秒。

3. 用户 linda 登录后，grep 命令的退出状态将变成 0，控制也将跳转到关键字 done 下面的那条语句。



范例 8-34

（脚本）

```
    #!/bin/sh
    # Scriptname: hour
1   hour=1
2   until [ $hour  -gt 24 ]
    do
3     case "$hour" in
      [0-9] |1[0-1])   echo  "Good morning!"
           ;;
      12)  echo  "Lunch time."
           ;;
      1[3-7])   echo  "Siesta time."
           ;;
      *)   echo  "Good night."
           ;;
      esac
4     hour=`expr $hour + 1`
5   done
```

（输出）

```
$ hour
Good morning!
Good morning!
    ...
Lunch time
Siesta time
    ...
Good night.
    ...
```

说明

1. 变量 hour 被初始化为 1。

2. test 命令测试 hour 的值是否大于 24。如果 hour 的值不大于 24，则进入循环体。也就是，当 until 后面那条命令返回一个非 0 的退出状态时，程序就进入 until 循环。循环将反复执行，直到条件为真。

3. case 命令计算变量 hour 的值，并测试每条 case 语句以寻找与其匹配的值。

4. 在控制返回循环顶部之前，变量 hour 被加 1。

5. done 命令标志循环体的结束。

8.6.5 循环控制命令

如果出现了某种情况，您可能需要跳出循环或返回循环顶部，或者需要某种办法来中断某个死循环。Bourne shell 提供了一组循环控制命令来处理这类情况。

shift 命令　指定参数时，shift 命令将参量列表左移指定的次数。没有给定参数时，shift 命令仅把参量列表左移一次。一旦列表被移动，左端那个参数就被永远地删除了。while 循环遍历位置参量列表时，经常会用到 shift 命令。

格式

```
shift [n]
```

范例 8-35

(无循环)
(脚本)

```
    #!/bin/sh
1   set joe mary tom sam
2   shift
3   echo $*
4   set `date`
5   echo  $*
6   shift 5
7   echo  $*
8   shift 2
```

(输出)

```
3   mary tom sam
5   Fri Sep 9 10:00:12 PDT 2004
7   2004
8   cannot shift
```

说明

1. 用 set 命令设置位置参量。$1 赋值为 joe，$2 赋值为 mary，$3 赋值为 tom，$4 则赋值为 sam。$*代表所有的位置参量。

2. shift 命令把位置参量左移一次，结果 joe 被移走了。

3. 打印移动之后的参量列表。

4. set 命令把位置参量重置为 UNIX 命令 date 的输出。

5. 打印新的参量列表。

6. 把参量列表左移 5 次。

7. 打印新的参量列表。

8. 由于左移的次数超过了列表中的参数个数，shell 向标准错误输出发送消息。

范例 8-36

(有循环)
(脚本)

```
    #!/bin/sh
    # Name: doit
    # Purpose: shift through command-line arguments
    # Usage: doit [args]
1    while [  $# -gt  0 ]
     do
2       echo  $*
3       shift
4   done
```

(命令行)

```
$ doit a b c d e
```



```
a b c d e
b c d e
c d e
d e
e
```

说明

1. while 命令测试一个数值表达式。如果位置参量的个数($#)大于 0，就进入循环体。来自命令行的位置参量作为实参。这个例子有 5 个位置参量。

2. 打印所有的位置参量。

3. 将参量列表左移一次。

4. 循环体到此结束，控制返回循环顶部。每进入一次循环，shift 命令都从参量列表中移走一个成员。第一次移动后，$#(位置参量的个数)变成 4。当$#减到 0 时，循环结束。

范例 8-37

(脚本)

```
    #!/bin/sh
    # Scriptname: dater
    # Purpose: set positional parameters with the set command
    # and shift through the parameters.
1   set `date`
2   while [ $# -gt 0 ]
    do
3       echo $1
4       shift
    done
```

(输出)

```
$ dater
Sat
Oct
16
12:12:13
PDT
2004
```

说明

1. set 命令获取 date 命令的输出，然后将其分别赋给$1~$6 这 6 个位置参量。

2. while 命令测试位置参量的个数是否大于 0。如果大于 0，就进入循环体。

3. echo 命令显示$1(第一个位置参量)的值。

4. shift 命令把参量列表左移一次。每一次循环都会将列表左移，直至列表为空。那时，$#将变成 0，循环也将终止。

break 命令　内置命令 break 用于强行退出循环，但不退出程序(退出程序要用 exit 命令)。执行 break 命令后，控制从关键字 done 的下一行开始。break 命令使控制从最内层循环退出来，因此，如果有嵌套循环，可以给 break 命令带一个数字作为参数，这样就能指定跳到哪个外层循环的外面。如果嵌套了 3 层循环，最外层循环就是第 3 层循环，中间那

层是第 2 层循环，最里面那层则是第 1 层循环。break 在退出无限循环时很有用。

格式

```
break [n]
```

范例 8-38

```
    #!/bin/sh
1   while true; do
2       echo Are you ready to move on\?
        read answer
3       if [ "$answer" = Y -o "$answer" = y ]
        Then
4          break
5       else
           ...commands...
        if
6   done
7   print "Here we are"
```

说明

1. true 命令是一个 UNIX 命令，它总是以状态 0 退出。true 命令常被用来启动无限循环。只要有分号分隔，就可以把 do 语句和 while 命令写在同一行上。由于 true 返回真，控制进入循环体。

2. 要求用户输入，把用户的输入赋给变量 answer。

3. 如果$answer 的值是 Y 或 y，控制便转向第 4 行。

4. 执行 break 命令，退出循环，控制转向第 7 行。打印一行文本："Here we are"。除非用户回答 Y 或 y，否则程序将继续要求用户输入。循环会无止境地进行下去。

5. 如果第 3 行的测试为假，就执行 else 命令。当循环体在关键字 done 处终止时，控制将从第一行的 while 顶部重新开始。

6. 循环体的结尾。

7. 执行 break 命令后，控制由此开始。

continue 命令 continue 命令在条件为真时，把控制转回循环的顶部。continue 下面的所有命令都被忽略。如果被嵌套在多层循环之内，continue 将控制转回最内层循环的顶部。如果给它一个数字作为参数，continue 就能将控制转到任一层循环的起点。假定嵌套了 3 层循环，最外层循环就是第 3 层循环，中间那层是第 2 层循环，最里面那层则是第 1 层循环[4]。

格式

```
continue [n]
```

范例 8-39

(邮件列表)

④ 如果给 continue 命令的数字参数大于循环的层数，就退出整个循环。



```
$ cat mail_list
ernie
john
richard
melanie
greg
robin
(脚本)
    #!/bin/sh
    # Scriptname: mailem
    # Purpose: To send a list
1   for name in 'cat mail_list'
    do
2       if ["$name"="richard"] ; then
3           continue
        else
4           mail $name < memo
        fi
5   done
```

说明

1. 执行完命令替换'cat mail_list'后，for 循环开始遍历从文件 mail_list 中得到的名称列表(例如：ernie john richard melanie greg robin)。

2. 如果 name 的值等于 richard，就执行 continue 命令，控制转回到循环顶部计算循环表达式的地方。richard 此时已被移出列表，所以赋给变量 name 的是下一个用户，melanie。这里其实不必给字符串 richard 加引号，因为它是一个单词。但是，在 test 命令中，给等号运算符(=)后面的字符串加引号确实是个好习惯，因为当字符串包含的单词多于一个时，例如 richard jones，如果不给它加引号，test 命令就会生成报错信息：

```
test: unknown operator richard(不可识别的运算符 richard)
```

3. continue 命令将控制转回循环顶部，因而跳过循环体中剩余的所有命令。

4. 给列表中的所有用户(richard 除外)邮寄一份 memo 的副本。

5. 循环体的结尾。

8.6.6 嵌套循环和循环控制

使用嵌套循环时，可以给 break 和 continue 命令一个整型的数字参数，让控制可以从内层循环跳到外层循环。

范例 8-40

```
(脚本)
    #!/bin/sh
    # Scriptname: months
```

```
1   for month in Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
    do
2      for week in 1 2 3 4
       do
          echo -n "Processing the month of $month. Okay?"
          read ans
3         if["$ans"=n -o -z "$ans"]
          then
4           continue 2
          else
            echo -n "Process week $week of $month?"
            read ans
            if ["$ans"=n -o -z "$ans"]
            then
5              continue
            else
               echo "Now processing week $week of $month."
               sleep 1
                     #Commands go here
               echo "Done processing..."
               fi
           fi
6      done
7   done
(输出)
$ months
Processing the month of Jan. Okay?
Processing the month of Feb. Okay? y
Process week 1 of Feb? y
Now processing week 1 of Feb.
Done processing..
Processing the month of Feb. Okay? y
Process week 2 of Feb? y
Now processing week 2 of Feb.
Done processing...
Processing the month of Feb. Okay? n
Processing the month of Mar. Okay? n
Processing the month of Apr. Okay? n
Processing the month of May. Okay? n
```

说明

1. 启动外层的 for 循环。执行第一轮循环时，把 Jan 赋给变量 month。

2. 启动内层的 for 循环。执行第一轮循环时，把 1 赋给变量 week。返回外层循环之前，要将内层循环完整地遍历一遍。

3. 如果用户输入字母 n 或按下回车键，就执行第 4 行。

4. continue 命令的参数是 2，所以将控制转回从内往外数的第 2 层循环(本例中是最外层)。没有参数的 continue 命令则把控制转回最内层循环的顶部。



5. 控制返回内层 for 循环的顶部。
6. 这个 done 终止最内层循环。
7. 这个 done 终止最外层循环。

8.6.7 I/O 重定向和子 shell

shell 可以通过管道或重定向将输入从文件传到循环，也可以通过管道或重定向将输出从循环传到文件。shell 将启动一个子 shell 来处理 I/O 重定向和管道。循环结束后，在循环中定义的所有变量对脚本的其余部分都是不可识别的。

将循环的输出重定向到一个文件 范例 8-41，范例 8-41 演示了如何将循环的输出重定向到一个文件中。

范例 8-41

```
(命令行)
1    $ cat memo
     abc
     def
     ghi
(脚本)
     #!/bin/sh
     # Program name: numberit
     # Put line numbers on all lines of memo
2    if [ $# -lt  1 ]
     then
3       echo  "Usage: $0 filename " >&2
        exit 1
     fi
4    count=1                      # Initialize count
5    cat $1 | while read line     # Input is coming from file on command line
     do
6        [ $count -eq 1 ] && echo "Processing file $1..." > /dev/tty
7        echo $count $line
8        count=`expr $count + 1`
9    done > tmp$$                 # Output is going to a temporary file
10   mv tmp$$ $1
(命令行)
11   $ numberit memo
     Processing file memo...
12   $ cat memo
     1 abc
     2 def
     3 ghi
```

说明

1. 显示文件 memo 的内容。

2. 如果用户运行该脚本时未提供命令行参数，参数个数($#)就会小于 1，于是显示报错信息。

3. 如果参数个数小于 1，就把 Usage 的消息发到 stderr(标准错误输出)。

4. 变量 count 被赋值为 1。

5. UNIX 的 cat 命令显示由$1 指定的文件的内容，cat 命令的输出被管道传给 while 循环。执行第一轮循环时，赋给 read 命令的是文件的第一行，执行下一轮循环时则赋给它文件的第二行，以后各轮以此类推。如果读输入成功，read 命令返回的退出状态是 0，失败时退出状态是 1。

6. 如果 count 的值是 1，就执行 echo 命令，把它的输出发往/dev/tty，即屏幕。

7. echo 命令打印 count 的值，后跟文件中的这行内容。

8. count 的值加 1。

9. 整个循环的输出，即$1 所指定的文件中的每一行，被重定向到文件 tmp$$，文件的第一行是个例外，它被重定向到终端，即/dev/tty。[5]

10. 文件 tmp$$被更名为$1 中指定的名称。

11. 执行该程序。所处理的文件名为 memo。

12. 执行完脚本后，显示文件 memo 的内容，可以看到每一行前面都加上了它的行号。

范例 8-42

```
(输入文件)
$ cat testing
apples
pears
peaches
(脚本 )
    #!/bin/sh
    # This program demonstrates the scope of variables when
    # assigned within loops where the looping command uses
    # redirection. A subshell is started when the loop uses
    # redirection, making all variables created within the loop
    # local to the shell where the loop is being executed.
1   while read line
    do
2       echo $line    # This line will be redirected to outfile
3       name=JOE
4   done < testing > outfile   # Redirection of input and output
5   echo Hi there $name
(输出)
5   Hi there
```

说明

1. 如果 read 命令的退出状态为真，就进入 while 循环的循环体。read 命令将从文件 testing 中读取输入，这是在第 4 行的 done 后面指定的。每执行一轮循环，read 命令都从文件 testing 中读取新的一行。

2. 该行输出被重定向到第 4 行中所示的 outfile 文件。

⑤ $$展开后是当前 shell 的 PID 号。把这个数字添在文件名的后面，可以确保文件名不重复。



3. 变量 name 被赋值为 JOE。由于该循环使用了重定向，因此这个变量是循环的局部变量。

4. 关键字 done 这行的内容包括从文件 testing 重定向输入，以及把输出重定向到文件 outfile。该循环的所有输出都被写到文件 outfile 中。

5. 在循环外，变量 name 是未定义的。它是 while 循环的局部变量，只在循环体内可识别。因为变量 name 没有值，所以只显示出字符串“Hi there”。

把循环的输出通过管道传给 UNIX 命令　循环的输出可以通过管道传给另一条(组)命令，也可以重定向到一个文件。

范例 8-43

```
(脚本)
    #!/bin/sh
1   for i in 7 9 2 3 4  5
2   do
       echo  $i
3   done | sort -n
(输出)
2
3
4
5
7
9
```

说明

1. for 循环遍历一组未排序的整数。

2. 在循环体中，脚本打印这组整数。输出将通过管道传给 UNIX 的 sort 命令，并按数值进行排序。

3. 在关键字 done 之后生成管道。在子 shell 中执行循环。

8.6.8　在后台执行循环

循环可以在后台执行。这样，程序可以继续运行而不必花时间等待循环完成处理。

范例 8-44

```
(脚本)
    #!/bin/sh
1   for person in bob jim joe sam
    do
2      mail $person < memo
3   done &
```

说明

1. for 循环要移走词表中的所有名字，包括：bob、jim、joe 和 sam。它将每个名字依次赋给变量 person。

2. 在循环体中，将向每个人发送一份 memo 的副件。

3. 关键字 done 后的符号&使得循环在后台运行。这样，当循环执行时，程序也同时继续向下执行。

8.6.9 exec 命令和循环

使用 exec 命令，不需要创建子 shell，就能打开或关闭标准输入和标准输出。这样，如果启动循环，循环体内创建的所有变量在循环终止后都还能保留。如果在循环中用了重定向，则循环内创建的所有变量都会消失。

exec 命令常被用来打开文件(根据文件名或文件描述符)以供读写。注意，文件描述符 0、1 和 2 已预留给标准输入、标准输出和标准错误输出。文件打开后，将得到下一个可用的文件描述符。例如，如果下个可用的描述符是 3，则新打开的文件分配到的文件描述符就是 3。

范例 8-45

```
(文件)
1    $ cat tmp
     apples
     pears
     bananas
     pleaches
     plums
(脚本)
     #!/bin/sh
     # Scriptname: speller
     # Purpose: Check and fix spelling errors in a file
2    exec < tmp                  # Opens the tmp file
3    while read line             # Read from the tmp file
     do
4        echo $line
5        echo -n "Is this word correct? [Y/N] "
6        read answer < /dev/tty  # Read from the terminal
7        case "$answer" in
8       [Yy]*)
9            continue;;
        *)
            echo "What is the correct spelling? "
10           read word < /dev/tty
11           sed "s/$line/$word/g" tmp > error
12           mv error tmp
13           echo $line has been changed to $word.
        esac
14   done
```

说明

1. 显示文件 tmp 的内容。
2. exec 命令改变标准输入(文件描述符 0)，使输入不再来自键盘，而是来自文件 tmp。
3. 启动 while 循环。read 命令从文件 tmp 读取一行输入。
4. 显示保存在变量 line 中的值。



5. 询问用户这个词的拼写是否正确。

6. read 命令从终端，即/dev/tty，读取用户的答复。如果不重定向为从终端直接获取输入，read 命令将继续从文件 tmp 读取输入，且此时文件 tmp 依然保持打开状态以等待读操作。

7. case 命令计算用户给出的答案。

8. 如果变量 answer 的值是以字母 Y 或 y 开头的一个字符串，就执行下一行的 continue 语句。

9. continue 语句使程序控制转回到第 3 行，即 while 循环的起点。

10. 再一次请求用户输入(该单词的正确拼写)。从终端(/dev/null)读取输入。

11. 无论 line 的值出现在文件 tmp 的哪个位置，sed 命令都把它替换为变量 word 的值，然后把输出写到文件 error 中。

12. 把文件 error 更名为 tmp，从而用文件 error 的内容覆盖文件 tmp 的原有内容。

13. 显示该行，从而可以看出所作的改动。

14. 关键字 done 标志循环体的结束。

8.6.10 IFS 和循环

shell 的内部字段分隔符(internal field separator，IFS)的值包括空格、制表符和换行符。IFS 被那些需要分析词列表的命令(如 read、set 和 for)用作词(token)分隔符。如果列表使用其他不同的分隔符，用户也可以重新设置 IFS。改变 IFS 的值之前，最好先把它的初始值保存到另一个变量中。这样，一旦需要，就可以很方便地把 IFS 恢复为默认值。

范例 8-46

(脚本)
```
   #/bin/sh
   # Scriptname: runit
   # IFS is the internal field separator and defaults to
   # spaces, tabs, and newlines.
   # In this script it is changed to a colon.
1  names=Tom:Dick:Harry:John
2  OLDIFS="$IFS"    # Save the original value of IFS
3  IFS=":"
4  for persons in $names
   do
5     echo  Hi $persons
   done
6  IFS="$OLDIFS"              # Reset the IFS to old value
7  set Jill Jane Jolene       # Set positional parameters
8  for girl in $*
   do
9     echo Howdy $girl
   done
```
(输出)
```
5  Hi Tom
   Hi Dick
```

```
    Hi Harry
    Hi John
9   Howdy Jill
    Howdy Jane
    Howdy Jolene
```

说明

1. 变量 names 被设置为字符串“Tom:Dick:Harry:John”。各个词之间用冒号分隔。

2. 把 IFS 的值(空白符)赋给另一个变量 OLDIFS。因为 IFS 的值是空白符，所以必须对它加引号进行保存。

3. 将 IFS 设置为冒号。现在，冒号被用作分隔符。

4. 变量替换完成后，for 循环以冒号作为词之间的内部字段分隔符，并依次对每个名字进行处理。

5. 显示词表中的所有名字。

6. 把 IFS 重新设成保存在 OLDIFS 中的初始值。

7. 设置位置参量，把$1 设为 Jill，把$2 设为 Jane，$3 则被设为 Jolene。

8. $*的值是所有位置参量，即：Jill、Jane 和 Jolene。for 循环在每次遍历时，会逐一将这些名字赋给变量 girl。

9. 显示参数列表中的每一个名字。

8.7 函数

函数是从 AT&T 的 UNIX System VR2 开始被引入 Bourne shell 的。函数其实就是一个名称，代表某条或某组命令。函数可以模块化程序、提高效率，甚至可以被保存在单独的文件中，需要时再被载入脚本。

接下来让我们回顾一些关于如何使用函数的重要规则。

(1) 将由 Bourne shell 来判断您使用的究竟是内置命令、函数，还是磁盘上的可执行程序。它先在内置命令中查找，然后是函数，最后才是可执行程序。

(2) 函数必须先定义，后使用。

(3) 函数在当前环境中运行，它可以共享所在脚本中的变量，还允许您以给位置参量赋值的方式向函数传递参数。如果在函数中使用 exit 命令，就会退出整个脚本。但是，如果函数的输入或输出被重定向，或者函数被括在反引号中(命令替换)，shell 就会创建一个子 shell，函数和它的变量以及当前工作目录都只能在子 shell 中被识别。函数退出后，在函数中设置的所有变量都会丢失，您将回到调用函数之前所在的目录。退出函数后，您将回到脚本中函数调用时的位置。

(4) return 语句返回函数执行的最后一条命令的退出状态，或者返回传给它的参数，返回值不能超过 255。

(5) 函数只能存在于定义它的 shell 中，不能向子 shell 输出。可以用 dot 命令来执行文件中保存的函数。



(6) 要列出函数和函数的定义，可以使用 set 命令。

(7) 陷阱(trap)和变量一样，被不同的函数共用。它们被脚本和脚本调用的函数共享。如果在函数中定义了一个陷阱，这个陷阱也会被脚本共享。但这种机制可能导致令人生厌的副作用。

(8) 如果函数存在另一个文件中，可以用 dot 命令把它们载入当前脚本。

格式

```
函数名() { 命令; 命令; }
```

范例 8-47

```
dir () { echo "Directories: " ; ls -l | nawk '/^d/ {print $NF}' ; }
```

说明

函数的名称是 dir。函数名后那对空的圆括号只是函数命名的必要语法，没有其他的目的。键入 dir 时，shell 将执行花括号里的命令。这个函数的功能是只列出当前工作目录下的子目录。注意，花括号两侧的空格是必需的。

8.7.1 清除函数

从内存中删除某个函数，应该使用 unset 命令。

格式

```
unset 函数名
```

8.7.2 函数的参数和返回值

由于函数是在当前 shell 中执行的，所以变量对函数和 shell 都是可见的。任何对函数中的环境所做的修改也会体现在 shell 中。您可以使用位置参量向函数传递参数。且位置参量是函数私有的，也就是说，函数对参数的操作不会影响在函数外使用的任何位置参量。

return 命令可用来退出函数并将控制转回程序调用函数的位置(记住，在脚本的任何位置使用 exit，包括在函数内，都会终止脚本的运行)。如果没有特别指定参数，函数的返回值其实就是函数中最后一条命令的退出状态。如果给 return 命令赋一个值，该值就被保存在变量?中，这个值可以是一个 0~255 之间的整数。因为规定了 return 命令只能返回 0~255 之间的整数，所以您可以用命令替换来获取函数的输出。具体做法与获取 UNIX 命令的输出时所做的一样：把整个函数放在一对反引号之间，然后将结果赋给某个变量。

范例 8-48

```
(使用 return 命令)
(脚本)
    #!/bin/sh
    # Scriptname: do_increment
1    increment () {
2       sum=`expr $1 + 1`
3       return $sum    # Return the value of sum to the script.
```

```
    }
4   echo -n "The sum is "
5   increment 5   # Call function increment; pass 5 as a parameter.
                  # 5 becomes $1 for the increment function.
6   echo $?       # The return value is stored in $?
7   echo $sum     # The variable "sum" is known to the function,
                  # and is also known to the main script.
(输出)
4,6   The sum is 6
7     6
```

说明

1. 定义函数 increment。

2. 调用函数时，第一个参数的值(即$1)加 1。相加的结果被赋给变量 sum。

3. 如果指定了参数，内置命令 return 将返回到主脚本中函数被调用时所在位置的下一行，同时将它的参数保存在变量？中。

4. 把这个字符串回显在屏幕上。

5. 用 5 作为参数调用 increment 函数。

6. 函数返回时，它的退出状态被保存在变量？中。如果 return 语句没有特定参数，函数的退出状态就是函数中最后一条命令的退出值。return 命令的参数必须是一个 0~255 之间的整数。

7. 变量 sum 虽然是在函数 increment 中定义的，但它的作用域却是全局的，因而可以在调用函数的脚本中被识别。这一行用于显示 sum 的值。

范例 8-49

```
(使用命令替换)
(脚本)
    #!/bin/sh
    # Scriptname: do_square
1   function square {
      sq=`expr $1 \* $1`
      echo "Number to be squared is $1."
2     echo "The result is $sq "
    }
3   echo "Give me a number to square. "
    read number
4   value_returned=`square $number`    # Command substitution
5   echo $value_returned
(输出)
3   Give me a number to square.
    10
5   Number to be squared is 10. The result is 100
```

说明

1. 定义函数 square。被调用时，它的功能是将它的参数(即$1)自乘。

2. 打印将该数自乘所得的结果。



3. 要求用户输入一个整数。主程序从这一行开始运行。

4. 用(用户输入的)一个整数作为参数来调用函数 square。因为函数被括在反引号之间，所以 shell 执行命令替换。函数的输出，即两条 echo 语句的输出，被赋值给变量 value_returned。

5. 命令替换去除了字符串“Number to be squared is 10.”和字符串“The result is 100”之间的换行符。

8.7.3 函数与 dot 命令

保存函数 函数常常被定义在.profile 文件中，这样，用户登入系统时，函数就被自动定义。函数不能被导出，但可以保存到一个文件中。所以，当需要使用某个函数时，只要用文件名作为参数，使用 dot 命令来调用该文件中定义的函数就可以了。

范例 8-50

```
1   $ cat myfunctions
2   go() {   # This file contains two functions
       cd $HOME/bin/prog
     PS1='`pwd` > '
       ls
    }
3   greetings() { echo "Hi $1! Welcome to my world." ; }
4   $ . myfunctions
5   $ greetings george
    Hi george! Welcome to my world.
```

说明

1. 显示文件 myfunctions 的内容，其中包含两个函数定义。
2. 定义的第 1 个函数为 go，它的功能是将主提示符设置为当前工作目录。
3. 定义的第 2 个函数为 greetings，它将问候由参数指定的用户。
4. dot 命令把文件 myfunctions 的内容加载到 shell 的内存空间。现在两个函数都在当前 shell 中定义了。
5. 调用并执行 greetings 函数。

范例 8-51

```
(下面的 .dbfunctions 文件中包含了主函数中使用的函数)
1   $ cat .dbfunctions
2   addon () {    # Function is named and defined in file .dbfunctions
3      while true
       do
          echo "Adding information "
          echo "Type the full name of employee "
          read name
          echo "Type address for employee "
          read address
          echo "Type start date for employee (4/10/88 ) : "
```

```
          read startdate
          echo $name:$address:$startdate
          echo -n "Is this correct? "
          read ans
          case "$ans"  in
          [Yy]*)
               echo "Adding info..."
               echo $name:$address:$startdate>>datafile
            sort -u datafile -o datafile
            echo -n "Do you want to go back to the main menu? "
            read ans
            if [ $ans = Y -o $ans = y ]
            then
4               return        # Return to calling program
             else
5               continue      # Go to the top of the loop
             fi
             ;;
          *)
             echo "Do you want to try again? "
             read answer
          case "$answer" in
          [Yy]*) continue;;
          *) exit;;
          esac
             ;;
      esac
   done
6  }  # End of function definition
--------------------------------------------------------------
```

(脚本)

```
7   #!/bin/sh
    # Scriptname: mainprog
    # This is the main script that will call the function, addon

    datafile=$HOME/bourne/datafile
8   . .dbfunctions    # The dot command reads the dbfunctions file into memory
    if [ ! -f $datafile ]
    then
       echo "`basename $datafile` does not exist" 1>&2
       exit 1
    fi
9   echo "Select one: "
    cat <<EOF
       [1] Add info
       [2] Delete info
       [3] Exit
    EOF
    read choice
    case "$choice" in
10     1)    addon        # Calling the addon function
             ;;
```



```
        2)    delete          # Calling the delete function
               ;;
        3)    update
               ;;
        4)
              echo Bye
              exit 0
               ;;
        *)    echo Bad choice
              exit 2
               ;;
    esac
    echo Returned from function call
    echo The name is $name
    # Variables set in the function are known in this shell.
```

说明

1. 显示文件.dbfunctions的内容。该文件并不是一个脚本，它只包含了函数的定义。

2. 定义函数addon。它的功能是往文件datafile中添加新的信息。

3. 进入while循环。如果循环体中没有类似break或continue的循环控制语句，它就会永远循环下去。

4. return命令把控制发送回程序中调用函数的位置。

5. 控制返回到while循环的顶部。

6. 右花括号结束函数定义。

7. 这是主脚本。该脚本中会用到函数addon。

8. dot命令将文件.dbfunction载入程序的内存空间。现在主脚本中已经有了函数addon的定义，可以使用它了。这样做的效果与直接在主脚本中定义函数一样。

9. 用here文档显示一个菜单，要求用户选择一个菜单项。

10. 调用addon函数。

8.8 捕获信号

如果您在程序运行时按下Ctrl+C或Ctrl+\组合键，程序就会在信号到达的那一刻立即终止。有时候您可能不想让程序在信号到达后立即终止。您可以安排程序忽略信号，继续运行或者先执行某些清理操作再退出脚本。trap命令使您能够控制程序收到信号后的行为。

信号被定义为由一个整数构成的异步消息，它可以由某个进程发给另一个进程，也可以在用户按下某些特定的键或发生某种异常事件时，由操作系统发给某个进程。trap命令[6]通知shell：如果收到某个信号，就终止正在执行的命令。如果trap命令后面跟着括在引号中的命令，则收到指定信号时，shell会执行这个命令串。shell要把这个命令串读两遍，设置信号陷阱时读一遍，信号到达时再读一遍。如果命令串两端是双引号，shell会在第一次

⑥ Morris I. Bolsky 和 David G. Korn 编著的 *The New KornShell Command and Programming Language*。

设置该陷阱时执行该命令串中所有的变量替换和命令替换。如果命令串两端是单引号，则其中的变量替换和命令替换要等到探测到并捕获信号后才进行。

使用命令 kill -l 就能得到所有信号的列表。表 8-5 提供了信号编号及相应的信号名称。

表 8-5 信号及其编号[⑦,⑧]

1) HUP	12) SYS	23) POLL
2) INT	13) PIPE	24) XCPU
3) QUIT	14) ALRM	25) XFSZ
4) ILL	15) TERM	26) VTALRM
5) TRAP	16) URG	27) PROF
6) IOT	17) STOP	28) WINCH
7) EMT	18) TSTP	29) LOST
8) FPE	19) CONT	30) USR1
9) KILL	20) CHLD	31) USR2
10) BUS	21) TTIN	
11) SEGV	22) TTOU	

格式

```
trap '命令; 命令' 信号编号
```

范例 8-52

```
trap 'rm tmp*; exit 1' 1 2 15
```

说明

当信号 1(挂起)、信号 2(中断)和信号 15(软件终止)中任何一个到达时，删除所有临时文件，然后退出。

如果脚本在运行过程中收到中断信号，则可以使用 trap 命令来选择不同的处理方式：正常处理该信号(默认方式)、忽略该信号或创建一个用以处理信号的可调用函数。

8.8.1 重置信号

要将信号重置为默认行为，在 trap 命令后面跟上信号的名称或编号即可。

范例 8-53

```
trap 2
```

说明

重置信号 2(INT 信号)的默认行为。这个信号用于终止进程，其作用等同于按下 Ctrl+C 组合键。

⑦ 因 OS/或 shell 输出的不同，kill 命令的输出也会不同。

⑧ 如果要了解全部的 UNIX 信号及其含义，可以登录 www.cybermagician.co.uk/technet/unixsignals.htm。有关 LINUX 信号的信息，可以登录 www.comptechdoc.org/os/linux/programming/linux_pgsignals.html。



范例 8-54

```
trap 'trap 2' 2
```

说明

把信号 2(INT)的默认动作设置为：当信号到达时，执行引号中的命令串。用户必须连按两次 Ctrl+C 组合键才能终止程序。第一个 trap 用于捕捉信号，第二个 trap 将该信号的陷阱重新设置为它的默认动作，即终止进程。

8.8.2 忽略信号

如果 trap 命令后面跟的是一对空引号，其后所列的信号都将被进程忽略。

范例 8-55

```
trap " " 1 2
```

说明

信号 1 和信号 2 将被 shell 忽略。

8.8.3 列出陷阱

键入 trap 就能列出所有陷阱和指定给它们的命令。

范例 8-56

(脚本)

```
    #/bin/sh
    # Scriptname: trapping
    # Script to illustrate the trap command and signals
1   trap 'echo "Ctrl-C will not terminate $0."' 2
2   trap 'echo "Ctrl-\ will not terminate $0."' 3
3   echo "Enter any string after the prompt."
    echo "When you are ready to exit, type \"stop\"."
4   while true
    do
        echo -n "Go ahead...> "
        read reply
5       if [ "$reply" = stop ]
        then
6           break
        fi
7   done
```

(输出)

```
Enter any string after the prompt.
When you are ready to exit, type "stop".
Go ahead...> this is it^C
Ctrl-C will not terminate trapping.
Go ahead...> this is never it ^\
Ctrl-\ will not terminate trapping.
Go ahead...> stop
$
```

说明

1. 第一个 trap 命令捕捉 INT 信号，即按下 Ctrl+C 组合键。如果用户在程序运行过程中按下 Ctrl+C 组合键，程序将执行引号中的命令。程序不会异常终止，而是会输出“Ctrl-C will not terminate trapping”，并且继续提示用户输入。

2. 第二个 trap 命令将在用户按下 Ctrl+\组合键(发出 QUIT 信号)时被执行。程序将显示字符串“Ctrl - \ will not terminate trapping”，然后继续运行。QUIT 信号的默认动作是终止进程并生成 core 文件。

3. 提示用户输入。

4. 进入 while 循环并显示提示“Go ahead…>”。

5. 将用户输入的内容赋给变量 reply，如果它的值等于 stop，就退出循环并终止程序。除非用 kill 命令终止进程，否则，我们只能用这种方法退出该程序。

6. break 命令使得控制从循环体中退出，从第 7 行之后开始。本例中，程序在第 7 行便结束了。

7. while 循环的结尾。

8.8.4 函数中的信号陷阱

如果您在函数中使用陷阱来处理某个信号，一旦该函数被调用，就会影响到整个脚本对该信号的处理。这个陷阱对脚本而言是全局的。下面这个例子中，陷阱被设置为忽略中断键(其作用等同于按下 Ctrl+C 组合键)。只有用 kill 命令终止该脚本才能跳出循环。这个例子说明在函数中使用陷阱可能会引起潜在的负作用。

范例 8-57

(脚本)

```
    #!/bin/sh
1   trapper () {
        echo "In trapper"
2       trap 'echo "Caught in a trap!"' 2
            # Once set, this trap affects the entire script. Anytime
            # ^C is entered, the script will ignore it.
    }
3   while :
    do
        echo "In the main script"
4       trapper
5       echo "Still in main"
        sleep 5
    done
```

(输出)

```
In the main script
In trapper
Still in main
^CCaught in a trap!
In the main script
In trapper
```




```
Still in main
^CCaught in a trap!
In the main script
```

说明

1. 定义 trapper 函数。该函数中定义的全部变量和陷阱对脚本都是全局性的。

2. trap 命令忽略信号 2，即中断键(Ctrl+C 组合键)。如果用户按下 Ctrl+C 组合键，程序会输出消息“Caught in a trap!”，但脚本将永远执行下去。

3. 主脚本启动了一个无限循环。

4. trapper 函数被调用。

5. 函数调用结束后，程序由此开始执行。

8.8.5 调试

使用 sh 命令的-n 选项，就能够对脚本进行语法检查而无需实际运行其中的任何一条命令。如果脚本中存在语法错误，shell 会将它显示出来；如果没有错误，shell 就不显示任何内容。

最常用的脚本调试手段是用-x 选项运行 set 命令，或把-x 选项作为参数传给 sh 命令。参见表 8-6 列出的调试选项。这些选项使您能够跟踪脚本的执行。此时，shell 对脚本中每条命令的处理过程是：先执行替换，然后显示，最后再执行它。shell 显示脚本中的行时，会在行首添上一个加号(+)。

表 8-6 调试选项

命　令	选　项	功　能
sh –x 脚本名	回显	在变量替换之后、执行命令之前，显示脚本的每一行
sh –v 脚本名	详细	在执行之前，按输入的原样输出脚本中各行
sh –n 脚本名	不执行	解释但不执行命令
set –x	打开回显	跟踪脚本的执行
set +x	关闭回显	关闭跟踪功能

如果打开详细选项，或者在启动 Bourne shell 时加上-v 选项(sh –v 脚本名)，脚本中的每一行都会按它在脚本中的原样显示在屏幕上，然后被执行。

范例 8-58

(脚本)
```
    #!/bin/sh
1   # Scriptname: todebug
    name="Joe Blow"
    if [ "$name" = "Joe Blow" ]
    then
       echo "Hi $name"
    fi
    num=1
```

```
        while [ $num -lt 5 ]
        do
            num=`expr $num + 1`
        done
        echo The grand total is $num
(命令行与输出)
2       $ sh -x todebug
        + name=Joe Blow
        + [ Joe Blow =  Joe Blow ]
        + echo Hi Joe Blow

        Hi Joe Blow
        num=1
        + [ 1 -lt 5 ]
        + expr 1 + 1
        num=2
        + [ 2 -lt 5 ]
        + expr 2 + 1
        num=3
        + [ 3 -lt 5 ]
        + expr 3 + 1
        num=4
        + [ 4 -lt 5 ]
        + expr 4 + 1
        num=5
        + [ 5 -lt 5 ]
        + echo The grand total is 5
        The grand total is 5
```

说明

1. 该脚本名为 todebug。您可以通过打开-x 选面来观察脚本的运行情况。每一次循环的结果都被显示在屏幕上，变量每次被设置和修改的值也被显示出来。

2. sh 命令用-x 选项启动 Bourne shell。回显选项被打开，脚本的每一行都被显示在屏幕上，行首添了个加号(+)。变量替换在显示之前就已执行。命令的执行结果将会出现在已显示的行的后面。

8.9 命令行

8.9.1 用 getopts 处理命令行选项

编写需要很多命令行选项的脚本时，位置参量不一定是最奏效。举个例子来说，UNIX 的 ls 命令使用了很多个命令行选项和参数(变量前面需要有一个短划线来引导，参数则不需要)。而选项可以通过多种方式传给程序，如：ls –laFi、ls –i –a –l –F、ls –ia –F 等。如果脚本需要参数，您可以为每个参数单独分配一个位置参量，如：ls –l –i –F，这 3 个以短划线引导的选项将被分别赋给$1、$2 和$3。但是，如果用户只用一个短划线选项列出所有的选



项，如 ls –liF，会出现什么情况呢？这时，-liF 将被整个赋给脚本中的$1。同样，getopt 函数也能使我们采用与 ls 程序相同的方法来处理选项和参数[9]。下面这个例子中，getopts 函数将允许 runit 程序处理以各种方式组合的参数。

范例 8-59

```
(命令行)
1    $ runit -x -n  200 filex

2    $ runit -xn200 filex

3    $ runit -xy

4    $ runit -yx -n 30

5    $ runit -n250 -xy file
(这些参数可以任意组合)
```

说明

1. 程序 runit 用了 4 个参数：x 是选项、n 也是选项，但它后面需要跟一个数字作为参数，filex 则是一个独立的参数。
2. 程序 runit 将选项 x、n 以及数字参数 200 组合在一起。filex 是一个参数。
3. 用 x 和 y 的选项组合调用程序 runit。
4. 调用程序 runit 时，选项 x 和 y 被组合在一起，选项 n 和数字参数 30 则被单独传递。
5. 调用程序 runit 时，将选项 n 和数字参数组合，选项 x 和 y 组合，filex 则单独传递。

在讨论程序 runit 的所有细节之前，我们先分析一下程序中调用 getopts 的那行语句，看看 geropts 是如何处理参数的。

范例 8-60

```
(runit 脚本中的一行)
while getopts :xyn: name
```

说明

- x、y 和 n 都是选项。选项就是那些从命令行传递给脚本的参数。例如：-x -y -n。
- 在命令行键入选项时前面要加一个短划线，如-x -y -z、-xyz、-x –yz 等。如果选项后面有冒号，则必需跟的是已命名的参数。比如-x filename，其中-x 是选项，filename 是已命名的参数。
- 遇到不含短划线的选项时，getopts 就认为选项列表已结束。
- 每次被调用时，getopts 都将找到的下一个选项去掉短划线，然后放到变量 name 中(这个变量名是任意的)。如果给的是非法变量，getopts 就会将 name 的值设为问号。
- OPTIND 是一个专用变量，它的初值是 1。getopts 每处理完一个命令行参数就将它加 1，变成 getopts 将要处理的下一个参数的编号。
- 若选项需带一个参数，则 getopts 命令将参数保存在 OPTARG 变量中。如果选项列

⑨ 请参见 UNIX 手册(第 3 节)中关于 C 库函数 getopt 的内容。

表的第 1 个字符是冒号，那么 shell 的变量 name 会被设为冒号，且变量 OPTARG 会被设置为合法选项的值。

getopts 脚本 下面这些例子将说明 getopts 如何处理参数。

范例 8-61

```
(脚本)
     #!/bin/sh
     # Program opts1
     # Using getopts -- First try --
1    while getopts xy options
     do
2         case $options in
3         x) echo "you entered -x as an option";;
          y) echo "you entered -y as an option";;
          esac
     done
(命令行)
4    $ opts1 -x
     you entered -x as an option
5    $ opts1 -xy
     you entered -x as an option
     you entered -y as an option
6    $ opts1 -y
     you entered -y as an option
7    $ opts1 -b
     opts1:  illegal option -- b
8    $ opts1 b
```

说明

1. getopts 命令被用作 while 命令的条件。getopts 命令后面列出了该程序的有效选项，即 x 和 y。循环体逐一测试每个选项。getopts 将去掉短划线的选项赋给变量 options。所有的参数都处理完之后，getopts 将以一个非零的状态退出，从而终止 while 循环。
2. case 命令用来测试在变量 options 中找到的每个可能的选项，即 x 或 y。
3. 如果某个选项是 x，则显示字符串 “You entered -x as an option”。
4. 在命令行给脚本 opts1 一个 x 选项，x 是一个合法选项，将被 getopts 处理。
5. 在命令行给脚本 opts1 一个 xy 选项，x 和 y 都是合法选项，都将被 getopts 处理。
6. 在命令行给脚本 opts1 一个 y 选项，y 是一个合法选项，将被 getopts 处理。
7. 给脚本 opts1 一个 b 选项，这是一个非法选项。getopts 将向 stderr 发送一条报错消息。
8. 前面没有短划线引导的选项不是所要处理的选项，于是 getopts 将停止处理参数。

范例 8-62

```
(脚本)
     #!/bin/sh
     # Program opts2
     # Using getopts -- Second try --
1    while getopts xy options 2> /dev/null
```



```
    do
2       case $options in
        x) echo "you entered -x as an option";;
        y) echo "you entered -y as an option";;
3       \?) echo "Only -x and -y are valid options"  1>&2;;
        esac
    done
```

(命令行)

```
    $ opts2 -x
    you entered -x as an option
    $ opts2 -y
    you entered -y as an option
    $ opts2 xy
    $ opts2 -xy
    you entered -x as an option
    you entered -y as an option
4   $ opts2 -g
    Only -x and -y are valid options
5   $ opts2 -c
    Only -x and -y are valid options
```

说明

1. 如果 getopts 发出报消息，将该消息重定向到/dev/null。

2. 遇到非法选项时，将会为 getopts 变量 options 赋一个问号。case 命令可用来检测问号，使得您能够在标准错误输出上显示自己的报错消息。

3. 如果变量 options 被赋值为问号，就执行这条 case 语句。反斜杠用来保护问号，这样 shell 就不会把问号当成自己的通配符而对它执行文件替换。

4. g 不是合法选项。因此变量 options 被赋值为问号，屏幕上显示报错消息。

5. c 不是合法选项。因此变量 options 被赋值为问号，屏幕上显示报错消息。

范例 8-63

(脚本)

```
    #!/bin/sh
    # Program opts3
    # Using getopts -- Third try --
1   while getopts dq: options
    do
       case $options in
2           d) echo "-d is a valid switch ";;
3           q) echo "The argument for -q is $OPTARG";;
            \?) echo "Usage:opts3 -dq filename ... " 1>&2;;
       esac
    done
```

(命令行)

```
4   $ opts3 -d
    -d is a valid switch
5   $ opts3 -q  foo
    The argument for -q is foo
```

```
6   $ opts3 -q
    Usage:opts3 -dq filename ...
7   $ opts3 -e
    Usage:opts3 -dq filename ...
8   $ opts3 e
```

说明

1. while 命令测试 getopts 的退出状态。如果 getopts 成功地处理了一个参数，它将返回退出状态 0，于是程序进入 while 循环的循环体。参数列表末尾的冒号表示 q 选项需要一个参数。这个参数将保存在专用变量 OPTARG 中。

2. d 是合法选项之一。如果把它作为一个选项输入，就将被保存在变量 options 中(不含短划线)。

3. q 是合法选项之一。q 选项需要一个参数。q 选项和它的参数之间必须有一个空格。q 和它的参数被传入脚本后，q 将被保存在变量 options 中((不带短划线)，它的参数则被保存在变量 OPTARG 中。如果 q 选项后未带参数，变量 options 中保存的将是一个问号。

4. d 选项是脚本 opts3 的合法选项。

5. 带参数的 q 选项是脚本 opts3 的合法选项。

6. 不带参数的 q 选项是错误的。

7. e 选项无效。如果选项非法，变量 options 中将保存一个问号。

8. 这个选项前面既没有短划线也没有加号。getopts 命令不会将其作为选项，它返回一个非零的退出状态。while 循环随之结束。

范例 8-64

(脚本)

```
    #!/bin/sh
    # Program opts4
    # Using getopts -- Fourth try --
1   while getopts xyz: arguments 2>/dev/null
    do
        case $arguments  in
2       x). echo "you entered -x as an option.";;
        y) echo "you entered -y as an option.";;
3       z) echo "you entered -z as an option."
            echo "\$OPTARG is $OPTARG.";;
4       \?) echo "Usage opts4 [-xy] [-z  argument]"
            exit 1;;
        esac
    done
5   echo "The initial value of \$OPTIND is 1.
        The final value of \$OPTIND is $OPTIND.
        Since this reflects the number of the next command-line argument,
        the number of arguments passed was `expr $OPTIND - 1`. "
```

(命令行)

```
$ opts4 -xyz foo
you entered -x as an option.
you entered -y as an option.
```




```
you entered -z as an option.
$OPTARG is foo.
The initial value of $OPTIND is 1.
The final value of $OPTIND is 3.
Since this reflects the number of the next command-line argument, the number
of arguments passed was 2.
$ opts4 -x -y -z  boo
you entered -x as an option.
you entered -y as an option.
you entered -z as an option.
$OPTARG is boo.
The initial value of $OPTIND is 1.
The final value of $OPTIND is 5.
Since this reflects the number of the next command-line argument, the number
of arguments passed was 4.
$ opts4 -d
Usage: opts4  [-xy] [-z argument]
```

说明

1. while 命令测试 getopts 的退出状态。如果 getopts 成功地处理了一个参数，它将返回退出状态 0，于是程序进入 while 循环的循环体。z 选项后面的冒号告知 getopts-z 选项必须带一个参数。如果某个选项要带参数，它的参数将保存在 getopts 的内置变量 OPTARG 中。

2. 如果 x 作为选项被传入脚本，将被保存在变量 arguments 中。

3. 如果 z 作为带参数的选项被传入脚本，它的参数将被保存在内置变量 OPTARG 中。

4. 如果传入的是无效选项，变量 arguments 中保存的将是一个问号，脚本将显示一条报错消息。

5. getopts 的专用变量 OPTIND 保存待处理的下一个选项的编号。它的值总是比实际的命令行参数个数多 1。

8.9.2 eval 命令和命令行解析

eval 命令处理命令行，先执行所有的 shell 替换，然后执行命令行。当平常的命令行解析无法满足要求时，就要使用 eval 命令进行第二次解析。

范例 8-65

```
1   $ set a b c d
2   $ echo The last argument is \$$#
3   The last argument is $4
4   $ eval echo The last argument is \$$#
    The last argument is d
5   $ set -x
    $ eval echo The last argument is \$$#
    + eval echo the last argument is $4
    + echo the last argument is d
    The last argument is d
```

说明

1. 设置 4 个位置参量。

2. 用户希望得到的结果是输出最后一个位置参量的值。\$输出一个美元符。$#的值是 4，即位置参量的个数。shell 求出$#的值后，不会再次解析命令行以得出$4 的值。

3. 输出的结果是$4，而不是最后那个参数。

4. shell 执行完所有的变量替换后，eval 命令又执行一次变量替换，然后执行 echo 命令。

5. 打开回显选项来观察解析的顺序。

范例 8-66

(摘自 SVR4 的 Shutdown 程序)

```
1    eval `/usr/bin/id | /usr/bin/sed 's/[^a-z0-9=].*//'`
2    if [ "${uid:=0}" -ne 0 ]
     then
3        echo $0: Only root can run $0
         exit 2
     fi
```

说明

1. 这个例子有一定难度。id 程序的输出将被发送给 sed，以提取出字符串中的 uid 部分。id 的输出结果是：

```
uid=9496(ellie) gid=40 groups=40
uid=0(root) gid=1(daemon) groups=1(daemon)
```

sed 命令中正则表达式的含义是：找出任何一个非字母、数字或等号的字符，删除该字符和它后面的所有字符。结果是将第一个左圆括号到行尾的所有内容替换为空。这样处理后所剩的内容就是 uid=9496 或 uid=0。

eval 完成对命令行的转换后，将执行所得的命令：

```
uid=9496
```

或

```
uid=0
```

例如，如果用户的 ID 是 root，eval 执行的命令将是 uid=0。这条命令将在脚本中创建一个称作 uid 的局部变量，并将它赋值为 0。

2. 测试变量 uid 的值是否为 0，其中用到了命令修饰符。

3. 如果 uid 的值不为 0，echo 命令显示脚本名($0)和这条消息。

8.10 shell 的调用选项

用 sh 命令启动 shell 时，可以通过指定选项来改变它的运行方式。参见表 8-7。



表 8-7 shell 的调用选项

选　项	含　义
-i	shell 在交互模式下运行。忽略 QUIT 和 INTERRUPT 信号
-s	从标准输入读取命令，将输出结果发送到标准错误输出
-c 字符串	从字符串中读取命令

8.10.1 set 命令和选项

除了处理命令行参数外，set 命令还可用来打开和关闭 shell 的选项。要打开某个选项，就在选项前加个短划线(-)。要关闭某个选项，则可在选项前加上加号(+)。参见表 8-8 中列出的 set 选项。

表 8-8 set 命令的选项

选　项	含　义
-a	标出所有被修改或输出的变量
-e	如果命令返回的状态非 0，则退出程序
-f	禁止 globbing(即文件名扩展)
-h	在定义函数时定位并记住函数所用命令，而不只是在执行函数命令时进行
-k	把所有的关键字参数放到命令的环境中，而不是只放命令名前面那些
-n	读命令但不执行，用于调试
-t	读取并执行完命令就退出
-u	在执行替换时，将未设置的变量视为错误
-v	按读入的原样打印 shell 输入行，用于调试。
-x	在执行命令时打印命令和它们的参数，用于调试
--	不改变任何标志

范例 8-67

```
1   $ set -f
2   $ echo *
    *
3   $ echo ??
    ??
4   $ set +f
```

说明

1. 打开 f 选项，禁止文件名扩展。
2. 星号没有被扩展。
3. 问号没有被扩展。
4. 关闭 f 选项，允许文件名扩展。

8.10.2 shell 的内置命令

shell 有很多内置在源代码中的命令。这些命令是内置的，所以 shell 不必到磁盘上定位它们，因此执行速度加快。表 8-9 列出了这些内置命令。

表 8-9 内置命令

命 令	功 能
:	空命令：返回退出状态 0
. file	dot 命令从文件 file 中读取命令并执行
break [n]	参见 8.6.5 节中的 break 命令
continue [n]	参见 8.6.5 节中的 continue 命令
cd	改变目录
echo [参数]	回显参数
eval 命令	shell 在执行命令前扫描命令行两次
exec 命令	运行命令，替换掉当前 shell
exit [n]	以状态 n 退出 shell
export [变量]	使变量可被子 shell 识别
hash	控制用于快速命令查找的内部哈希表
kill [-信号 进程]	向由 PID 号或作业号指定的进程发送信号。请参见/usr/include/sys/signal.h 中的信号列表
getopts	被 shell 脚本用来解析命令行并检查合法的选项
login [用户名]	登录系统
newgrp [参数]	改变用户的组 ID，从而将其分配到一个新的用户组
pwd	打印出当前的工作目录
read [var]	从标准输入读取一行，保存到变量 var 中
readonly [var]	将变量 var 设为只读，不允许重置该变量
return [n]	从函数中退出，n 是传递给 return 命令的退出状态值
set	请参见表 8.18
shift [n]	将位置参量左移 n 次
stop pid	暂停第 pid 号进程的运行
suspend	终止当前 shell 的运行(对登录 shell 无效)
time	打印所有当前 shell 启动的进程所用的累计用户时间和系统时间
trap [arg] [n]	当 shell 收到信号 n(n 为 0、1、2 或 15)时，执行 arg
type [命令]	打印出命令的类型，例如：pwd 是 shell 的一个内置变量，在 ksh 中则是命令 whence –v 的别名
umask [八进制数]	用户文件关于属主、属组和其他用户的创建模式掩码
unset [名字]	取消指定变量或函数的值
wait [pid#n]	等待 PID 号为 n 的后台进程结束，并报告它的结束状态
ulimit [选项 大小]	设置进程可用资源的最大限额
umask [掩码]	如果不带参数，则打印出关于文件权限的文件创建掩码



习题 8 Bourne shell 入门

1. 哪个进程把登录提示符输出到屏幕上？
2. 哪个进程为 HOME、LOGNAME 和 PATH 赋值？
3. 怎么才能知道自己正在运行哪种 shell？
4. 在哪里(哪个文件)指定登录 shell？
5. 解释/etc/profile 和.profile 这两个文件之间的区别。shell 先执行哪一个？
6. 编辑您的.profile 文件，完成下列功能：
   a) 欢迎用户。
   b) 如果路径中不包括您的主目录，将其加入。
   c) 用 stty 命令设置退格键的删除功能。
   d) 键入：.profile
      dot 命令的功能是什么？

习题 9 元字符

1. 创建一个名为 wildcards 的目录。cd 到该目录后在命令行键入：

```
touch ab abc a1 a2 a3 all a12 ba ba.1 ba.2 filex filey AbC ABC ABc2 abc
```

2. 写出能实现下列功能的命令，并测试所写的命令：
   a) 列出所有名称以 a 开头的文件。
   b) 列出所有名称以至少一个数字结尾的文件。
   c) 列出所有名称以 a 或 A 开头的文件。
   d) 列出所有名称以句号跟一个数字结尾的文件。
   e) 列出所有名称中包含字母 a 的文件。
   f) 列出所有名称由 3 个字母组成，且所有字母都大写的文件。
   g) 列出所有名称以 10、11 或 12 结尾的文件。
   h) 列出所有名称以 x 或 y 结尾的文件。
   i) 列出所有名称以数字、大写字母或小写字母结尾的文件。
   j) 列出所有名称不是以 a、b 或 B 开头的文件。
   k) 删除名称为两个字符，且以 a 或 A 开头的文件。

习题 10 重定向

1. 与终端关联的 3 个文件流的名称是什么？
2. 什么是文件描述符？
3. 给出完成下列任务的命令：
   a) 把 ls 命令的输出重定向到文件 lsfile。
   b) 重定向 date 命令的输出，将其追加到文件 lsfile 尾部。
   c) 把 who 命令的输出重定向到文件 lsfile，会出现什么结果？
4. 执行下列操作。
   a) 只输入 cp，会出现什么结果？
   b) 把上面这个例子产生的报错消息保存到一个文件中。

c) 用 find 命令从父目录开始，找出所有类型为目录的文件。把标准输出保存到文件 found 中，把所有报错消息保存到文件 found.errs 中。
d) 获取 3 个命令的输出，将它们重定向到文件 gottem_all 中。
e) 使用管道来运行 ps 和 wc 命令，查出您正在运行的进程数目。

习题 11 第 1 个脚本

1. 写一个名为 greetme 的脚本。
   a) 包含一段注释，列出您的姓名、脚本的名称和写这个脚本的目的。
   b) 问候用户。
   c) 输出当前日期和时间。
   d) 输出这个月的日历。
   e) 输出您的机器名。
   f) 输出当前这个操作系统(cat /etc/motd)的名称和版本。
   g) 输出父目录中所有文件的列表。
   h) 输出 root 正在运行的所有进程。
   i) 输出变量 TERM、PATH 和 HOME 的值。
   j) 输出磁盘的使用情况(du)。
   k) 用 id 命令输出您的组 ID。
   i) 输出“Please，could you loan me $50.00?”
   m) 跟用户说“Good bye”并且显示当前时间(请参考 date 命令的手册页)。
2. 确保您的脚本可执行。

```
chomd +x greetme
```

3. 您的脚本中第一行是什么？为什么需要写这一行？

习题 12 命令行参数

1. 写一个名为 rename 的脚本，这个脚本需要两个参数：第一个参数是文件的原名，第 2 个参数则是文件的新名称。如果用户没有提供两个参数，就在屏幕上显示一条信息，然后退出脚本。下面是说明该脚本如何工作的一个例子：

```
$ rename
Usage: rename oldfilename newfilename
$

$ rename file1 file2
file1 has been renamed file2
Here is a listing of the directory:
a file2
b file.bak
```

2. 下面这个 find 命令(SunOS)将列出根分区上所有大于 100KB、并且在上周被修改过的文件(可查看当前系统上的帮助信息，以确定 find 的正确语法)。

```
find / -xdev -mtime -7 -size +200 -print
```



3. 写一个名为 bigfiles 的脚本。这个脚本带两个参数：一个是 mtime 的值，另一个则是 size 的值。如果用户没有提供这两个参数，就向 stderr 发送一条相应的报错消息。

4. 如果有时间，写一个名为 vib 的脚本，用它来为 vi 创建备份文件。备份文件的名称由原始文件的名称加上后缀.bak 组成。

习题 13 获取用户的输入

1. 写一个名为 nosy 的脚本，该脚本将执行下列操作：

a) 询问用户的全名——名和姓。

b) 用用户名问候他(她)。

c) 询问用户的出生年份，并计算出他(她)的年龄(使用 expr 命令)。

d) 询问用户的登录名，并输出他(她)的用户 ID(从/etc/passwd 中获得)。

e) 告诉用户他(她)的主目录在哪。

f) 向用户显示他(她)正在运行的进程。

g) 告诉用户现在是星期几，并且用 12 小时制的时间格式告诉他(她)现在的时间。输出结果形式为：

```
The day of the week is Tuesday and the current time is 04:07:38 PM.
```

2. 创建一个名为 datafile 的文本文件(除非已存在了这个文件)。文件中的每条记录都包含若干由冒号分隔的字段。记录包含的字段是：

a) 名和姓

b) 电话号码

c) 地址

d) 出生日期

e) 工资

3. 创建一个名为 lookup 的脚本。

a) 包含一节注释，指明脚本名、作者姓名、时间和编写这个脚本的目的。编写这个脚本的目的是要显示已排序 datafile 的内容。

b) 按姓氏对 datafile 排序。

c) 向用户显示 datafile 的内容。

d) 告诉用户文件中一共有多少条记录。

4. 尝试用-x 和-v 选项来调试您的脚本。这些命令的使用方法是什么？它们有何不同？

习题 14 条件语句

1. 写一个名为 checking 的脚本来执行如下操作：

a) 接收一个命令行参数：用户的登录名。

b) 检查用户是否提供了命令行参数。

c) 检查对应的用户是否在/etc/passwd 文件中，如果在，就输出：

```
Found <user> in the /etc/passwd file.
```

d) 若不在，则输出：

```
No such user on our system.
```

2. 在脚本 lookup 中，询问用户是否要往文件 datafile 中增加一条记录。如果用户回答 yes 或 y，则：

a) 提示用户输入姓名、电话号码、地址、出生日期和工资。将每一项分别保存在一个单独的变量中。在字段间加入冒号，然后把这条信息追加到文件 datafile 中。

b) 按姓氏对该文件排序。告诉用户这条记录已被加入，向他(她)显示该行，并在行首标出行号。

习题 15 条件语句与文件测试

1. 改写 checking。检查完指定的用户是否在/etc/passwd 文件中之后，程序接着检查这个用户是否已登录系统。如果是，程序就输出正在运行的所有进程；否则，程序将告诉用户：

```
<所指定的用户> is not logged on.
```

2. 脚本 lookup 的运行需要依靠 datafile 文件。在脚本 lookup 中，检查文件 datafile 是否存在，是否可读并且可写。在脚本 lookup 中增加一个如下所示的菜单：

```
[1] Add entry
[2] Delete entry
[3] View entry
[4] Exit
```

a) 您已经在脚本中完成了增加记录(Add entry)的编写。现在要在增加记录的程序中增加代码，以检查用户提供的姓名是否已在文件 datafile 中出现，如果是，就告诉用户，否则，则增加这条新记录。

b) 现在编写删除记录(Delete entry)、查看记录(View entry)和退出(Exit)函数。

c) 脚本中处理删除的部分应该首先检查记录是否存在，然后才去删除它。如果记录不存在，则通知用户这个错误。否则，就删除这条记录，并且告诉用户记录已被删除。退出时，一定要用一个数字来代表相应的退出状态。

d) 您如何从命令行检查退出状态？

习题 16 case 语句

1. BSD UNIX 上的 ps 命令与 AT&T UNIX 上的有所不同。在 AT&T UNIX 上，列出所有进程的命令是：

```
ps -ef
```

而 BSD UNIX 上对应的命令则是：

```
ps -aux
```

请编写一个名为 systype 的程序，用它来检查多种不同的系统类型。所要测试的系统将包括：

```
AIX
Darwin (Mac OS X)
Free BSD
```




```
HP-UX
IRIX
Linux
OS
OSF1
SCO
SunOS(Solaris/SunOS)
ULTRIX
```

Solaris、HP-UX、SCO 和 IRIX 是 AT&T 一类的系统，其余的则是 BSD 风格的系统。

您正在使用的 UNIX/Linux 的版本信息将被输出到 stdout。系统的名称可以用 uname –s 命令或从文件/etc/motd 中获取。

习题 17　循环

任选一题：

1. 写一个名为 mchecker 的脚本，用来检查是否有新邮件到达，如果有新邮件，则向屏幕显示一条消息。

程序将获取用户的邮件假脱机文件的长度(AT&T 类系统上，邮件假脱机文件位于/usr/mail/$LOGNAME 中，UCB 类系统上，假脱机文件的位置是在/usr/spool/mail/$USER 中。如果您找不到它们，可以使用 find 命令)。这个脚本将连续循环进行，每 30 秒一次。每一轮循环时，脚本都将邮件假脱机文件的长度与上一轮循环时获取的长度进行比较。如果新的长度大于旧的长度，就在屏幕上显示一条消息：Username，You have new mail.

文件的长度可以从 ls –l、wc –c 的输出中或使用 find 命令找到。

2. 写一个名为 dusage 的脚本程序，向一组用户逐一发送邮件，告诉用户他(她)当前已用的磁盘块数目。用户的名单保存在文件 potential_hogs 中。potential_hogs 文件列出的用户之一是 admin。

a. 用文件测试检查文件 potential_hogs 是否存在并可读。

b. 用循环遍历整个用户名单。只向磁盘用量超过 500 块的用户发送邮件。跳过用户 admin，即不向他(她)发邮件。邮件的信息保存在 dusage 脚本的 here 文档中。

c. 保存一份收到邮件的用户的名单。通过创建日志文件来完成这一任务。给名单上的所有用户都发送完邮件后，输出收到邮件的用户人数和名单。

习题 18　函数

1. 将习题 16 中的 systype 程序改写为一个返回系统名的函数。在程序 checking 中使用 ps 命令，调用该函数来确定应使用哪些选项。

在 AT&T UNIX 系统上，列出所有进程的 ps 命令：

```
ps -ef
```

在 BSD UNIX 系统上，对应的 ps 命令：

```
ps -aux
```

2. 写一个名为 cleanup 的函数，它将删除所有临时文件并退出脚本。如果程序运行过程中收到中断或挂起信号，trap 命令将调用 cleanup 函数。

3. 用 here 文档在脚本 lookup 中增加一个如下所示的菜单：

```
[1] Add entry
[2] Delete entry
[3] Change entry
[4] View entry
[5] Exit
```

为每个菜单项编写一个处理函数。当用户选择一个有效菜单项之后，程序完成相应的函数操作，然后询问用户是否需要再次查看菜单。如果用户输入的菜单项无效，则程序应该输出：

```
Invalid entry, try again.
```

然后重新显示菜单。

4. 在 lookup 脚本的菜单项 View entry 下创建一个子菜单。询问用户是否要查看所选用户的详细信息：

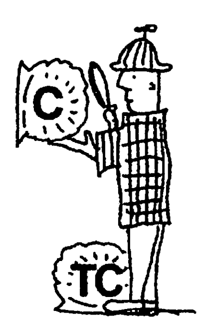
```
a) Phone
b) Address
c) Birthday
d) Salary
```

5. 在脚本中使用 trap 命令，使程序在运行过程中收到中断信号时，执行清除操作。



chapter





交互式 C shell 与 TC shell

9.1 简介

交互式 shell 是指该 shell 的标准输入、标准输出和标准错误输出都与终端相连。C shell(csh)和 TC shell(tcsh)提供了许多 Bourne shell 中所没有的实用特性以增强其交互性，包括文件名自动补全、命令别名、历史替换、作业控制等。作为编程语言来讲，这两种 shell 大同小异，但是 TC shell 相对其前驱 C shell，在交互性方面有了更加丰富的内容。本节涵盖了 C shell 和 TC shell 常用的一些交互式特性。如果需要的是一些诸如好用的命令行快捷方式等 TC shell 所特有的增强交互特性，请参阅本章的后半部分，从 9.13 节，“交互式 TC shell 的新特性”开始。

C/TC shell 启动

在 shell 显示提示符之前，有几个进程会先执行(参见图 9-1)，系统引导后，第一个运行的进程是 init，它的进程标识符(PID)是 1。init 进程从文件 inittab 中读取指令(System V 的做法)，或者派生一个 getty 进程(BSD 的做法)。这些进程负责打开终端端口，提供标准输入(stdin)的来源，以及标准输出(stdout)与标准错误输出(stderr)的去处，并且在屏幕上显示一个登录提示符。接下来系统将执行/bin/login 程序。login 程序将依次执行下面这些工作：提示用户输入口令、加密并验证用户输入的口令、设置初始工作环境并初始化 shell，C shell 运行的是/bin/csh 程序，TC shell 运行的则是 bin/tcsh 程序。

当 shell 登录时，会产生一系列的调用：C/TC shell 在/etc 目录下搜索启动文件，如果存在就首先执行。例如，C shell 的启动文件是/etc/csh.cshrc 和/etc/csh.login(参见第 16 章“系统管理员与 shell”)。然后 C/TC shell 分别在用户主目录查找.cshrc 文件和.tcshrc 文件。这两个初始化文件用于定制用户工作的 C/TC shell 环境。执行完.cshrc 或.tcshrc 文件中的命令以后，接下来会执行.login 文件中的命令。不同的是，每次启动一个新的 C/TC shell，.cshrc 或.tcshrc 文件都会执行，而.login 文件，它也包含用于初始化用户环境的命令和变量，仅在用户登录时执行一次。执行完这些文件中的命令，屏幕上将显示一个提示符，说明此时 C/TC



shell 正在等待用户命令。如今大多数系统都基于以上步骤建立了一个图形用户界面。当使用诸如 CDE 之类的桌面工具时，将会看到图标和菜单。如果从菜单中选择终端(terminal)，可以看到 shell 提示符，此时就可以键入命令了。

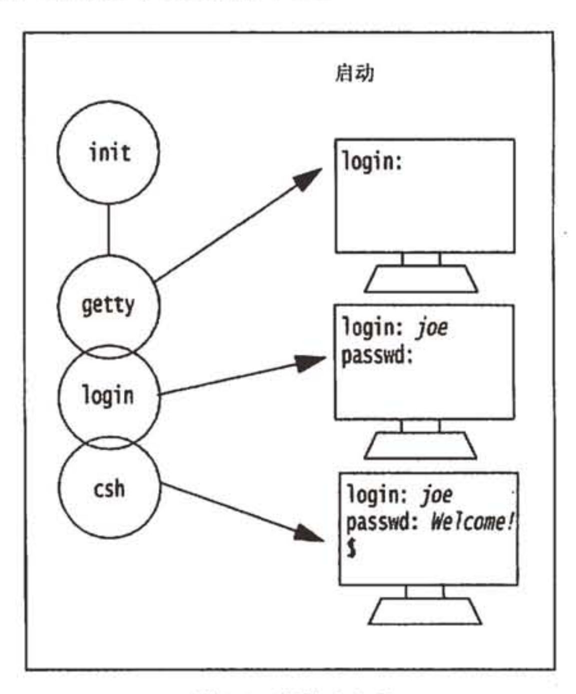


图 9-1 启动 C shell

9.2 环境

9.2.1 初始化文件

csh/tcsh 程序启动后，首先要分别执行用户主目录下的两个文件：.cshrc/.tcshrc 和.login。这些文件使用户能够初始化他们自己的环境。

.cshrc 文件与.tcshrc 文件 cshrc 文件中包含 C shell 的变量设置，每次登录和启动 csh 的子 shell 时这个文件都会被执行。同样，.tcshrc 文件包含 TC shell 的变量设置，每次登录和启动 tcsh 的子 shell 时这个文件也会被执行。这些文件设置的内容相同。例如，别名和历史通常是在这里设置的。

范例 9-1

```
( .cshrc 文件)
1   if ( $?prompt ) then
2        set prompt = "\!  stardust > "
3        set history = 32
4        set savehist = 5
5        set noclobber
6        set filec fignore = ( .o )
```

```
7         set cdpath = ( /home/jody/ellie/bin /usr/local/bin /usr/bin )
8         set ignoreeof
9         alias m more
          alias status 'date;du -s'
          alias cd 'cd \!*;set prompt = "\! <$cwd> "'
      endif
```

说明

1. 如果已经设置了提示符($?prompt)，则 shell 是在交互式状态下运行的，而不是在脚本中运行。

2. 主提示符被设置为当前历史事件的数目、名字 stardust 和字符>，替换掉了默认的%提示符。

3. 变量 history 被设置为 32。该变量用于控制在屏幕上显示的历史事件的数目。当键入命令 history 时，屏幕上将显示您之前输入的最后 32 条命令(请参见 9.3.5 节“命令行历史”)。

4. 退出系统后，历史列表通常会被清空。变量 savehist 可用来指定保存历史列表尾部命令的数目。本例中，最后 5 条命令将保存在主目录下的.history 文件中，因此，当您再次登录进入系统时，shell 将检查该文件是否存在，并将所保存的历史命令放在新的历史列表顶部。

5. 设置变量 noclobber，以防止用户在使用重定向时因疏忽而删除文件。例如，sort myfile >myfile 将覆盖原有的 myfile 文件。如果设置了变量 noclobber，此时屏幕上就会显示消息：file exists(文件已存在)。

6. 变量 filec 用于文件名自动补全，只需键入文件名的头几个字符，然后按下 ESC 键，shell 就会补全文件名的剩余部分。输入文件名时，如果按下^D(即 Ctrl+D 组合键)，C shell 就会列出匹配这个字符串的所有文件。变量 fignore 则允许您将不愿受文件名补全功能影响的文件排除在外。这个例子中，即使设置了 filec，所有以.o 结尾的文件名(目标文件)也不会受 filec 的影响(请参见 9.8.7 节“文件名补全：变量 filec”)。

7. 变量 cdpath 的值被设为一组路径项。当转换目录时，如果只指定目录名，而这个目录又不是当前工作目录的直接子目录，shell 就会搜索 cdpath 中列出的目录项，并试着在其中某个位置找到指定的目录，然后转到该目录。

8. 变量 ignoreeof 将禁止用^D 退出系统。有些从键盘接收输入的 UNIX 工具是以^D 作为输入结束的标志(比如 mail 程序)。如果系统比较慢，用户会忍不住多按几次^D，这样就会出现意外的结果：第一次按下^D 使 mail 程序终止，第二次按下的^D 则使用户退出系统。如果设置了 ignoreeof，则用户必须键入 logout 才能退出系统。

9. 设置别名(aliase)的目的是给某条或某组命令一个简化的符号。别名设置好之后，用户键入别名时，shell 就会执行它对应的那条(组)命令。这个例子给 more 命令取的别名是 m，每次您输入 m 时，shell 就会执行 more 命令。status 这个别名会打印出当前日期和用户磁盘使用情况的摘要。另一个别名 cd 则在每次用户转换目录时生成一个新的提示符，这个新的提示符中包括当前历史事件的数目(\!*)和用< >括着的当前工作目录$cwd(参见 9.4 节“别名”)。



.login 文件 用户第一次登录时，shell 会将.login 文件执行一遍。这个文件中的内容一般是环境变量和终端的设置。窗口程序通常在这个文件中启动。环境变量可以被当前 shell 派生的进程继承，只需设置一次，而终端设置则不需要为每个进程重置，因此，这些设置属于.login 文件的内容。

范例 9-2

```
( .login 文件)
1   stty -istrip
2   stty erase ^h
3   stty kill ^u
    #
    # If possible start the windows system.
    # Give a user a chance to bail out
    #
4   if ( $TERM == "linux" ) then
5          echo "Starting X windows.  Press Ctrl-C \
               to exit within the next 5 seconds "
           sleep 5
6          startx
7   endif
8   set autologout=60
```

说明

1. stty 命令用于设置终端选项。如果使用了-istrip，输入的字符就不会被删去最高位而转换为 7 位。

2. stty 命令将退格键(即 Ctrl+H 组合键)设置为删除命令。

3. 所有以#开头的行都被当成注释，它们不是可执行语句。

4. 如果当前终端窗口(tty)是控制台(Linux)，则执行下一行命令；否则，程序控制转到最后那个 endif。

5. 这一行将回显在屏幕上。如果用户没有按下 Ctrl+C 组合键终止当前进程，则程序暂停 5 秒，然后启动 X 窗口程序。

6. 如果系统是 Linux，则 startx 程序将启动 X 窗口。

7. 这个 endif 标志着最内层 if 结构的结束。

8. autologout 变量被设置为 60，这样一旦系统不活跃达到 60 分钟，用户将自动注销(从登录 shell 中)。

9.2.2 搜索路径

shell 用变量 path 来定位用户在命令行键入的命令。查找自左向右进行。句点代表当前工作目录。如果在路径中列出的所有目录和当前工作目录下均未发现要找的命令，shell 就会往标准错误输出发送这样一条消息：Command not found。通常，推荐在.login[①]文件中设置路径。C/TC shell 中设置搜索路径的方式与 Bourne shell 和 Korn shell 有所不同，C/TC shell

① 不要将搜索路径变量与.cshrc 文件中设置的 cdpath 变量混淆。

中每条路径之间用空白符分隔：

```
set path = (/usr/bin /usr/ucb /bin /usr .)
echo $path
/usr/bin /usr/ucb /bin /usr .
```

环境变量 PATH 显示为：

```
echo $PATH
/usr/bin:/usr/ucb:/bin:.
```

C/TC shell 会在内部自动更新环境变量 PATH，因为从该 shell 中启动的某些其他程序(如 Bourne shell 和 Korn shell)可能需要使用变量 path，C/TC shell 要保持与它们的兼容。

9.2.3 rehash 命令

C shell 建立了一个内部哈希表，该表由搜索路径所列目录的内容组成。如果搜索路径中不含句点，则句点目录(即当前工作目录)中的文件不会被放入这个哈希表。为了提高效率，shell 将使用这个哈希表来查找用户在命令行键入的命令，而不是每次都去搜索路径中查找。如果在搜索路径列出的某个目录中增加了新的命令，就必须重新计算内部哈希表，实现的方法是：

```
% rehash
```

%是 C shell 的命令提示符。在命令提示符后改变自己的路径或启动另一个 shell 后，都会自动重新计算这个哈希表。

9.2.4 hashstat 命令

hashstat 命令显示一组性能数据，用以说明从哈希表查找命令的高效性。这组性能数据按“命中”和“未命中”的次数进行统计。如果您所使用的命令大部分是在路径尾部被找到，shell 就要比在路径前端找到大部分命令时多做不少工作，导致未命中次数高于命中次数。一旦出现这种情况，你可以将命中次数最多的目录移到路径的前端，以此来提高效率。

```
% haststat
2 hits, 13 misses, 13%②
```

9.2.5 source 命令

source 命令是 shell 的一个内置命令，即 shell 内部代码的一部分。它被用于执行一条命令或来自文件的一组命令。执行命令时，shell 通常会派生一个子进程来执行命令，以确保命令所做的任何改变都不会影响原来那个 shell(称作父 shell)。source 命令使得程序在当前 shell 中被执行，因此，该文件中定义的所有变量都将成为当前 shell 的环境的一部分。如果修改了.cshrc 或.login 文件，通常要用 source 命令重新执行它们。例如，如果在进入系统后修改了搜索路径，就可以输入命令：

---

② 在不包含 vfork 的系统中，这条命令将打印哈希记录的数量和大小。例如：1024 hash buckets of 16 bits each。



```
% source .login 或 % source .cshrc
```

9.2.6 shell 提示符

C shell 有两个提示符：主提示符是百分号(%)，次提示符则是问号(?)。TC shell 使用>作为其默认的主提示符(参见 9.13.2 节“Shell 提示符”关于 tcsh 增强的提示符设置)。用户登录进系统后，终端上显示的就是主提示符，以表明它正等待用户键入命令。主提示符可以被重置。当您在提示符后编写脚本时，如果需要使用 C/TC shell 的编程结构，例如，条件判断或循环，shell 就会显示次提示符，这样你就能在下一行继续。此提示符将在每个换行符后持续出现，直到编程结构正确结束为止。但次提示符不能被重置。

主提示符 交互式运行时，主提示符等待您键入命令并按下回车键。如果你不想使用默认的提示符，只需在.cshrc 文件中重新设置它，就会出在当前 shell 和它的所有子 C shell 中。如果你只想为当前登录会话框修改主提示符，则应在 shell 提示符后设置它。

范例 9-3

```
1  % set prompt = "$LOGNAME > "
2  ellie >
```

说明

1. 主提示符被设置为用户的登录名，后面还跟了符号>和一个空格。
2. 显示的是新的提示符。

次提示符 当您直接在提示符后编写脚本时，次提示符就会出现在屏幕上。只要你进入 shell 编程结构并按下回车键，次提示符就会出现，并且持续出现直到该编程结构正常结束为止。必须反复练习，才能直接在提示符后写出正确的脚本。输入命令后，一旦按下回车键，你就再也不能退回去修改了，而且，C shell 的历史记录机制不保存出现次提示符之后的命令。

范例 9-4

```
1  % foreach pal (joe tom ann)
2  ? mail $pal < memo
3  ? end
4  %
```

说明

1. 这是一个联机直接编写脚本的例子。C shell 认为在 foreach 循环后还应有更多的输入，因而显示出次提示符。foreach 循环将处理括号内列出的每一个词。

2. 第一轮循环中，joe 被赋给变量 pal。这条命令将 memo 的内容通过邮件发送给用户 joe。执行下一轮循环时，则轮到 tom 被赋给变量 pal，以此类推。

3. end 语句标志循环的结束。处理完括号中所列的全部数据项后，循环结束，shell 显示主提示符。

4. 显示主提示符。

9.3 C/TC shell 命令行

用户登录进系统后，C/TC shell 就在屏幕上显示出它的主提示符，默认分别为%和>。shell 其实就是命令解释器。交互式运行时，shell 从终端读取命令，将命令行分解为词。一个命令行由一或多个词(token)组成，词之间以空白符(空格或制表符)分隔，以换行符结束，换行符是通过按下回车键产生的。命令行的第一个词是要执行的命令，其后的词则是命令的选项或参数。这条命令可能是 UNIX/Linux 的一个可执行程序(例如 ls 或 pwd)，也可能是一个别名或内置命令(如 cd 或 jobs)，还可能是某个 shell 脚本。命令中可以包含称为元字符(metacharacters)的特殊字符，shell 分析命令行时必须解释元字符。如果命令行在换行符前面的最后一个字符是反斜杠。则可在下一行继续该行的内容[3]。

9.3.1 退出状态

命令和程序会在终止后向父进程返回一个退出状态(exit status)。退出状态是一个 0~255 之间的整数。依照惯例，如果程序退出时返回的状态是 0，则说明命令执行成功，退出状态非零，则说明命令因某种原因而运行失败。因此，C/TC shell 的状态变量被设置为它执行的最后一条命令的退出状态。程序运行的是成功还是失败，由它的编写者来决定。

范例 9-5

```
1   % grep "ellie" /etc/passwd
    ellie:GgMyBsSJavd16s:9496:40:Ellie Quigley:/home/jody/ellie
2   % echo $status
    0
3   % grep "nicky" /etc/passwd
4   % echo $status
    1
5   % grep "scott" /etc/passsswd
    grep: /etc/passsswd: No such file or directory
6   % echo $status
    2
```

说明

1. grep 程序在文件/etc/passwd 中查找模式 ellie，并成功。显示/etc/passwd 的各行。
2. 状态变量被设置为 grep 命令的退出值，0 表示命令执行成功。
3. grep 程序未能在文件/etc/passwd 中找到用户 nicky。
4. grep 程序没找到指定模式，因而返回退出状态 1。
5. 因为打不开文件/etc/passswd，grep 运行失败。
6. grep 程序没找到指定文件，因而返回退出状态 2。

9.3.2 命令编组

命令行可以由多条命令组成：各条命令之间用分号隔开，命令行以换行符结束。

---

③ 命令行的长度可以是 256 个字符或更长。其他版本的 UNIX 上这个值还可能更大。



范例 9-6

```
% ls; pwd; cal 2004
```

说明

C shell 从左往右依次执行命令，直至遇到换行符。

命令编组的另一个用处是将多条命令的输出结果集中起来，一同经管道发给另一命令或重定向到某个文件。shell 将在一个子 shell 中执行这些命令。

范例 9-7

```
1   % ( ls ; pwd; cal 2004 ) > outputfile
2   % pwd; ( cd / ; pwd ) ; pwd
    /home/jody/ellie
    /
    /home/jody/ellie
```

说明

1. 每条命令的输出都被发往文件 outputfile。如果没有加圆括号，前两个命令的输出将被发送到屏幕，只有 cal 命令的输出才被重定向到输出结果文件。

2. pwd 命令显示当前工作目录。圆括号使得其内部的命令在子 shell 中处理。cd 命令内置在 shell 中。在子 shell 中，目录被设置为 root，当前工作目录也显示在屏幕上。离开子 shell 后显示的则是原 shell 的当前工作目录。

9.3.3 命令的条件执行

对于条件执行，两个命令串由两个特殊的元字符——双与号和双竖号(&&和||)来分隔。这些字符右边的命令执行与否将由左边命令的退出条件来决定。

范例 9-8

```
% grep '^tom:' /etc/passwd && mail tom < letter
```

说明

如果第一条命令执行成功(其退出状态为 0)，则执行&&后的第 2 条命令。本例中，如果 grep 命令在 passwd 文件中成功地找到了 tom，就执行右边的命令。mail 命令会把文件 letter 的内容发送给 tom。

范例 9-9

```
% grep '^tom:' /etc/passwd || echo "tom is not a user here."
```

说明

如果第一条命令执行失败(其退出状态非 0)，则执行| |后的第 2 条命令。本例中，如果 grep 命令未能在 passwd 文件中找到 tom，就执行右边的命令。echo 程序将会在屏幕上显示：tom is not a user here。

9.3.4 后台命令

通常情况下，当执行命令时，命令程序是在前台运行的，命令执行结束后，提示符将重新出现。但是，等待命令完成有时并不合理。只要在命令行末尾加上一个与号(&)，shell 就会立即返回 shell 提示符，这样您就不必等到上一条命令结束后才开始执行新的命令。在后台运行的命令被称为后台作业(background job)，它会在运行过程中将输出发送到屏幕。如果两条命令同时往屏幕发送输出，结果就会混淆。为了避免这种情况，你可以让后台运行的作业将输出发往某个文件，或者用管道重定向到某个设备，例如打印机。一般在后台启动一个新的 shell 窗口是很方便的。这样，您就可以对两个窗口进行访问，一个是原有的，一个新启用的。

范例 9-10

```
1  % man xview | lp&
2  [1] 1557
3  %
```

说明

1. xview 程序的帮助页的输出被重定向到打印机。命令行末尾的与号用于将作业放在后台运行。

2. 有两个整数出现在屏幕上：方括号中的整数说明这是放在后台运行的第一个作业，第二个整数是这个作业的 PID 号。

3. 立即出现的 shell 提示符。当程序在后台运行时，shell 在前台提示您可以继续输入另一个命令。

9.3.5 命令行历史

C/TC shell 内置了历史机制(tcsh 对历史机制的增强特性请参见 9.14.2 节“TC shell 命令行历史”)。它将你在命令行键入的命令(称为历史事件，history events)保存为一个带编号的清单。你可以直接从历史列表中调出某条命令再次执行，而不必重新输入。命令行历史的替换字符是感叹号，常常被称作 bang 字符。内置命令 history 可用来显示历史清单。

范例 9-11

```
(命令行)
% history
1 cd
2 ls
3 more /etc/fstab
4 /etc/mount
5 sort index
6 vi index
```

说明

历史命令清单用于显示用户最近在命令行输入的命令。清单中每个事件的前面都对应一个编号。



设置历史 可以通过设置 C shell 的变量 history 来指定保存在历史清单中、显示在屏幕上的事件的数目。这个变量通常是在用户的初始化文件.cshrc 中设置。

范例 9-12

```
set history=50
```

说明

从终端输入的最近 50 条命令被保存起来，键入 history 命令就能将它们显示在屏幕上。

保存命令行历史 如果要跨不同的登录界面来保存历史事件，则应该设置 savehist 变量。这个变量通常在用户的初始化文件.cshrc 中被设置。

范例 9-13

```
set savehist=25
```

说明

命令历史清单中的最近 25 条命令被保存，当您再次登录进系统时，它们将出现在历史清单的顶部。

显示命令行历史 history 命令用于显示历史清单中的事件。这个命令还提供一些选项来控制所显示事件的数目和格式。事件的编号不必从 1 开始。如果您在命令历史清单中保存了 100 条命令，同时将变量 history 设置为 25，那么您将只看到清单中保存的最后 25 条命令(TC shell 支持使用方向键。参见 9.14.2 节中“访问历史文件中的命令”部分)。

范例 9-14

```
% history
1 ls
2 vi file1
3 df
4 ps -eaf
5 history
6 more /etc/passwd
7 cd
8 echo $USER
9 set
```

说明

历史清单被显示出来，其中的每条命令都编了号。

范例 9-15

```
% history -h          # Print without line numbers
ls
vi file1
df
ps -eaf
history
more /etc/passwd
cd
```

```
echo $USER
set
history -n
```

说明

不加行号显示命令历史清单。

范例 9-16

```
% history -r          # Print the history list in reverse
11 history -r
10 history -h
 9 set
 8 echo $USER
 7 cd
 6 more /etc/passwd
 5 history
 4 ps -eaf
 3 df
 2 vi file1
 1 ls
```

说明

逆序显示命令历史清单。

范例 9-17

```
% history 5        # Prints th last 5 events on the history list
7    echo $USER
8    cd
9    set
10   history -n
11   history 5
```

说明

显示命令历史清单中的最后 5 条命令。

重新执行命令 重新执行历史清单中的命令需要用到感叹号(bang)。如果您键入两个感叹号(!!)，shell 就会执行上一条命令。如果您只键入一个感叹号，接着键入一个数字，而这个数字在历史清单中又有关联的命令，shell 就会执行对应的那条命令。如果您键入一个感叹号和一个字母，则 shell 执行历史清单中最后一条以该字母开头的命令。脱字符(^)也可用作编辑前一条命令的快捷方式。

范例 9-18

```
1   % date
    Mon Apr 26  8 12:27:35 PST 2004
2   % !!
    date
    Mon Apr 26 12:28:25 PST 2004
3   % !3
    date
```



```
    Mon Apr 26 12:29:26 PST 2004
4   % !d
    date
    Mon Apr 26 12:30:09 PST 2004
5   % dare
    dare: Command not found.
6   % ^r^t
    date
    Mon Apr 26 12:33:25 PST 2004
```

说明

1. 在命令行执行 date 命令。历史清单随之被更新，date 成了清单中的最后一条命令。
2. !!(bang bang)从历史清单中取出最后那条命令，这条命令再次被执行。
3. 再次执行历史清单中第 3 条命令。
4. 再次执行命令清单中最后那条以字母 d 开头的命令。
5. 敲错了命令。
6. 用脱字符在历史清单的最后那条命令中替换字母。命令中第一个 r 被替换为 t。

范例 9-19

```
1   % cat file1 file2 file3
       <Contents of file1, file2, and file3 are displayed here>
    % vi !:1
    vi file1

2   % cat file1 file2 file3
       <Contents of file1, file2, and file3 are displayed here>
    % ls !:2
    ls file2
    file2

3   % cat file1 file2 file3
    % ls  !:3
    ls file3
    file3

4   % echo a b c
    a b c
    % echo !$
    echo c
    c

5   % echo a b c
    a b c
    % echo !^
    echo a
    a

6   % echo a b c
    a b c
    % echo !*
    echo a b c
```

```
    a b c

7   % !!:p
    echo a b c
```

说明

1. cat 命令把文件 file1、file2、file3 的内容显示到屏幕上。历史列表被更新，命令行被分解为若干个词，词的编号从 0 开始。如果在词的编号前加个冒号，就能将这个词从历史清单中提取出来。标记!:1 的含义是：取出历史清单中最后那条命令的第 1 个参数，替换掉命令串中的!:1。最后那条命令的第 1 个参数是 file1(编号 0 的单词即命令本身)。

2. !:2 被替换为上一条命令的第 2 个参数，即 file2，并且成为 ls 的参数。命令运行结果是打印出 file2(file2 是第 3 个参数)。

3. ls !:3 的含义是：找到历史清单中的最后一条命令，取出该命令中第 4 个词，把它作为参数传给 ls 命令(file3 是第 4 个词)。

4. 带美元符($)的感叹号(!)代表历史清单中最后那条命令的最后一个参数。此时这个参数是 c。

5. 脱字符(^)代表命令后的第 1 个参数。后跟脱字符的感叹号(!)代表历史清单中最后那条命令的第 1 个参数。本例中指的是 a。

6. 星号(*)代表命令后的所有参数。感叹号(!)后跟一个星号，代表历史清单中最后那条命令的所有参数。

7. 打印历史清单中最后一条命令，但不执行。历史清单被更新。现在你可以对该行执行脱字符替换。

9.4 别名

别名是 C/TC shell 中用户自定义的命令简写形式(关于 tcsh 别名机制的增强，参见 9.17 节“TC shell 别名”)。当某个命令要带很多选项和参数，或者命令语法很难记住时，别名就变得很有用。在命令行设置的别名不会被子 shell 继承。别名的设置通常在文件.cshrc 或.tcshrc 中进行。每个新 shell 启动时都要执行.cshrc 或.tcshrc 文件，所以，该文件中设置的所有别名都会为新 shell 重置。别名可以传递给 shell 脚本，但是这样会导致潜在的移植问题，除非所用的别名是直接定义在脚本中的。

9.4.1 列出别名

内置命令 alias 能够列出所有已设置的别名。输出时先打印别名，然后才是它所代表的实际的命令或命令集。

范例 9-20

```
% alias
co      compress
cp      cp -i
```



```
ls1       enscript -B -r -Porange -f Courier8 !* &
mailq     /usr/lib/sendmail -bp
mroe      more
mv        mv -i
rn        /usr/spool/news/bin/rn3
uc        uncompress
uu        uudecode
vg        vgrind -t -s11 !:1 | lpr -t
weekly    (cd /home/jody/ellie/activity; ./weekly_report; echo Done)
```

说明

alias 命令把命令的别名(简称)列在第 1 列，第 2 列中则是别名代表的实际命令。

9.4.2 创建别名

alias 命令也可用来创建别名：第一个参数是别名的名称，即命令的简称。该行的剩余部分就是单个命令或命令集，这些命令将在别名出现时被执行。多条命令之间用分号分隔，包含空格或元字符的命令必须用单引号引用。

范例 9-21

```
1   % alias m more
2   % alias mroe more
3   % alias lF 'ls -alF'
4   % alias cd  'cd \!*; set prompt = "$cwd  >"'
    % cd ..
    /home/jody >  cd /          # New prompt displayed
    / >
```

说明

1. 给 more 命令设了一个别名：m。
2. 把 more 命令的别名设为 mroe。这样能方便那些常拼写出错的人。
3. 由于命令中包含空白字符，所以加了单引号。LF 是命令 ls –alF 的别名。
4. 执行 cd 时，cd 的别名将使 cd 进到以变量命名的目录，然后设置提示符为当前工作目录后跟字符">"。别名使用!*的方法和它在历史机制中使用的方法相同。反斜杠符号将阻止历史机制在别名使用它前首先对!*求值。\!*可以代表历史清单中最近命令的变量。

9.4.3 删除别名

unalias 命令用来删除一个别名。如果要临时关闭一个别名，可以在别名的名称后加上一个反斜杠。

范例 9-22

```
1   % unalias mroe
2   % \cd ..
```

说明

1. unalias 命令从所定义的别名列表中删除别名 mroe。

2. 只在这次命令的执行中临时关闭别名 cd。

9.4.4 别名环

当一个别名定义引用了另一个别名，而这个别名又引用回最初的别名，就会出现别名环现象。

范例 9-23

```
1   % alias m more
2   % alias mroe m
3   % alias m mroe        # Causes a loop
4   % m datafile
    Alias loop.
```

说明

1. 将 m 指定为 more 的别名。每次使用 m 时，shell 将执行 more 命令。

2. mroe 定义为 m 的别名。如果键入 mroe，别名 m 将被调用，more 命令将被执行。

3. 这是一个错误。如果别名 m 被使用，它将调用别名 mroe, 别名 mroe 又引用回 m，会导致一个别名环。不过没什么坏结果，您只会得到一个错误信息。

4. 使用别名 m。这将陷入别名循环：m 调用 mroe，mroe 调用 m，然后 m 再调用 mroe，如此反复下去。当然，循环不会永远继续下去，C shell 将发现这个错误，并显示一个报错信息。

9.5 操作目录栈

在工作过程中，如果需要在目录树中相同的一些目录之间来回地转换，可以将这些目录压进一个目录栈，通过操作目录栈可以很方便地访问这些目录。目录栈可以形象地比作自助餐厅里的一摞盘子，每个盘子都摞在其他盘子的上面，第一个盘子处于底端。内置命令 pushd 会将目录入栈，popd 命令则将目录出栈(参见后面的例子)。目录栈指的是一个编号的目录列表，最近使用的目录放在栈顶。从栈顶的目录开始编号，将其编号为 0，接下来的目录编号为 1，以此类推。内置命令 dirs 加上-v 选项，将显示编号目录栈。

pushd 命令与 popd 命令

使用一个目录作为参数的 pushd 命令将这个新目录加入到目录栈并同时转换到这个目录。如果参数是一个长划线(-)，则它代表的是先前的工作目录。如果参数是一个+符号和一个数字(n)，则 pushd 从栈中取出第 n 个目录并将其压入栈顶，接着转换到该目录。如果没有参数，则 pushd 将目录栈栈顶的两个目录进行交换，这样就方便了目录的向前向后切换。有很多可以控制 pushd 的工作方式的 shell 变量(参见 9.10.2 节“局部变量”)。

如果要跨登录会话框保存目录栈，需要在 tcsh 初始化文件(例如~/.tcshrc)中设置 savedirs 变量。目录栈将被存储在文件~/.cshdirs 中，并且在 shell 启动时，自动对该文件执行 source 命令。



popd 命令将栈顶目录弹出，并转换至该目录。

表 9-1 是一个目录栈变量列表。

表 9-1　目录栈变量

变　量	功　能
deextract	如果设置，pushd +n 将取出目录栈第 n 个目录，然后将其压入栈顶
dirsfile	一个文件名，用于跨登录会话框存放目录栈
dirstack	用于显示栈内容或将目录赋给栈
dunique	在将一个目录压入栈之前，首先删除栈中所有同名的目录
pushdsilent	执行 pushd 时不打印目录栈
pushdtohome	如果设置，则无参数的 pushd 等价于 pushd ~或 cd
pushtohome	若不带参数，则压入用户主目录文件
savedirs	跨登录会话框保存目录

范例 9-24

```
1   % pwd
    /home/ellie

    % pushd ..
    /home ~

    % pwd
    /home

2   % pushd       # swap the two top directories on the stack
    ~ /home

    % pwd
    /home/ellie

3   % pushd perlclass
    ~/perlclass  ~  /home

4   % dirs -v
    0     ~/perlclass
    1     ~
    2     /home

5   % popd
    ~     /home
    % pwd
    /home/ellie

6   % popd
    /home
```

```
    % pwd
    /home

7   % popd
    popd: Directory stack empty.
```

说明

1. 第一个 pwd 命令显示当前工作目录：/home/ellie。下一条以..为参数的的 pushd 命令将父目录(..)压入栈中。pushd 命令的输出显示/home 是目录栈是栈顶目录，用户主目录(~)，也就是/home/ellie，则处于栈底。pushd 命令同时还把当前目录切换至刚刚压入栈的目录上，也就是/home。新目录通过 pwd 命令显示出来。
2. 不带参数的 pushd 交换栈顶的两个目录并切换到新的栈顶目录上。在本例中，是切换回用户主目录/home/ellie。
3. pushd 命令将其参数~/perlclass 入栈，并切换至该目录。
4. 内置的 dirs 命令显示编号的目录栈，编号 0 为最高。
5. popd 命令将栈顶目录弹出，并切换至该目录。
6. popd 命令将另一个目录弹出，并切换至该目录。
7. 因为目录栈已空，popd 命令无法再弹出更多的目录，于是它显示出上面的错误信息。

9.6 作业控制

作业控制是 C/TC shell 的一项强大功能，使得用户能够在后台或前台运行作业。通常，在命令行输入的命令都在前台运行，并且持续运行于前台直至结束。如果使用多个窗口，就不需要作业控制，因为只要另外打开一个窗口就可以启动新的任务。但是，如果只有一个终端，作业控制就将是一项非常有用的功能。表 9-2 列出了作业命令的清单(TC shell 对作业控制功能的增强参见 9.18 节“TC shell 作业控制”)。

表 9-2　作业控制命令

命　令	含　义
jobs	列出所有正在运行的作业
^Z(即 Ctrl+Z)	中止(暂停)作业，屏幕上将出现提示符
bg	启动被中止的后台作业
fg	将后台作业调到前台
kill	向指定作业发送 kill 信号

9.6.1 &号和后台作业

如果预知某个命令要花很长时间才能完成，不妨在命令后加上一个&号，让这个作业在后台执行。这样一来，C shell 提示符将立刻重现，您就可以键入下一条命令。此时将有两条命令同时在运行，一个在后台，一个在前台。这两条命令都将它们标准输出发送到屏



幕上。如果要把某个作业放到后台执行，您最好将它的输出重定向到某个文件或通过管道发送给某个设备(比如打印机)。

范例 9-25

```
1   % find . -name core -exec rm {} \; &
2   [1]  543
3   %
```

说明

1. find 命令在后台运行(如果没有-print 选项，find 命令就不会向屏幕发送任何输出)[4]。
2. 方括号中的数字表示这将是在后台运行的第一个作业。该进程的 PID 是 543。
3. 提示符立刻重现。shell 等待用户的输入。

9.6.2 暂停键序列和后台作业

要暂停某个程序，只要按下暂停键序列^Z(即 Ctrl+Z 组合键)。按下^Z 后，作业就会暂停(中止)，shell 提示符将重新出现，暂停的程序要等到用户发出 fg 或 bg 命令才能恢复运行(使用 vi 编辑器时，ZZ 命令用于写入和保存文件。不要把它与^Z 搞混，后者将暂停 vi 会话)。如果您在后台还有暂停的作业时执行注销操作，屏幕上就会出现这样一条消息："There are stopped jobs"。

9.6.3 jobs 命令

C/TC shell 的内置命令 jobs 显示出当前后台中正在运行或暂停的活动程序。正在运行的含义是作业正在后台执行。如果程序被中止，就进入暂停状态，即没有被执行。两种情形下，终端都是空闲的，可以接受其他的命令。

范例 9-26

(命令行)

```
1   % jobs
2   [1] + Stopped    vi filex
    [2] - Running    sleep 25

3   % jobs -l
    [1] + 355  Stopped     vi filex
    [2] - 356  Running     sleep 25

4   [2] Done sleep 25
```

说明

1. job 命令列出当前所有的活动作业。

2. 标记[1]是第一个作业的编号。加号表示该作业不是最近一个被放入后台的作业。短划线则表示这是最近一个被放入后台的作业。Stopped 表示该进程被用户用^Z 暂停，目前不处于活动状态。

---

④ find 命令的语法要求 exec 语句的结尾必须有个分号。在分号前面加上反斜杠是为了防止它被 shell 解释。

3. -l 选项(长清单)在显示作业号的同时也显示作业的 PID。标记[2]是第二个作业的编号，本例中，第二个作业是最后一个放入后台的作业。短划线表示这是最近的作业。这条 sleep 命令正在后台运行。

4. 作业将在 sleep 命令运行 25 秒后结束，屏幕上会出现一条消息说明该作业已结束。

9.6.4 前台和后台命令

fg 命令用于将后台作业调入前台。bg 命令则用于重新启动暂停的后台进程。如果要对某个特定作业进行控制，可以用百分号和作业编号作为 fg 和 bg 命令的参数。

范例 9-27

```
1   % jobs
2   [1] + Stopped    vi filex
    [2] - Running    cc prog.c -o prog

3   % fg %1
    vi filex
    (vi session starts)

4   % kill %2
    [2] Terminated   cc prog.c -o prog

5   % sleep 15
    (Press ^z)
    Stopped

6   % bg
    [1] sleep 15 &
    [1] Done    sleep 15
```

说明

1. jobs 命令列出当前正在运行的一些进程，这些进程被称为作业。
2. 第一个作业是一个 vi 会话，该作业已被中止。第二个作业是 cc 命令。
3. 将编号为[1]的作业调入前台。作业编号前面要有一个百分号。
4. kill 是内置命令。默认情况下，它向进程发送一个 TERM(终止)信号。kill 命令的参数可以是作业号或进程的 PID。
5. sleep 命令因用户按下^Z 而中止。现在这条 sleep 命令将不占用 CPU，而在后台被挂起。
6. bg 命令使得最后一个后台任务开始在后台执行。sleep 程序将在执行恢复之前开始按秒倒计时。

9.7 shell 元字符

元字符是一些特殊的字符，它们被用来代表不同于其本身的对象。既非数字又非字母



的字符就可能是元字符，这是一条经验法则。与 grep、sed 和 awk 这些程序类似，shell 有一套自己的元字符，通常被称为 shell 通配符(shell wildcards)[⑤]。shell 的元字符有多种用途，例如：组合多条命令、简写文件名和路径名、对输入/输出执行重定向和管道操作、将命令放入后台等。表 9-3 列出了部分 shell 元字符。

表 9-3 shell 的元字符

元字符	作　用	示　例	示例说明
$	变量替换	set name=Tom echo $name Tom	将变量 name 设为 Tom，并显示变量的值
!	历史替换	!3	重新执行历史清单中的第 3 个事件
*	文件名替换	rm *	删除所有的文件
?	文件名替换	ls ??	列出所有文件名为两个字符的文件
[ ]	文件名替换	cat f[123]	显示 f1、f2、f3 的内容
;	命令分隔符	ls;date;pwd	依次执行每个命令
&	后台运行	lp mbox&	打印在后台进行。立即返回提示符
>	输出重定向	ls >file	重定向标准输出到 file
<	输入重定向	ls <file	重定向到标准输入为 file
>&	输出和出错重定向	ls >&file	将输出和出错重定向到 file
>!	如果设置了 noclobber，忽略它	ls >!file	如果文件存在，覆盖它，即使设置了 noclobber
>>!	如果设置了 noclobber，忽略它	ls >>!file	如果文件不存在，创建它，即使设置了 nocobber
( )	将命令分组在一个子 shell 中执行	(ls ; pwd) >tmp	执行命令并将结果发送到 tmp 文件
{ }	将命令分组在这个 shell 中执行	{ cd /;echo 4cwd }	改为根目录并显示当前工作目录

9.8 文件名替换

解析命令行时，shell 会用元字符来简写与某个特定字符组相匹配的文件名或路径名。表 9-4 中列出的文件名替换元字符将会被扩展为一组按字母排序的文件名。将元字符扩展为文件名的过程也被称作 globbing。C shell 与其他几种 shell 不同，当不能将元字符扩展为指定的文件名时，它将报告“No match”。

⑤ grep、sed 和 awk 之类的程序都有一组称作正则表达式的元字符，它们用于模式匹配。不要将这些元字符与 shell 的元字符搞混。

表 9-4　Shell 元字符与文件名替换

元　字　符	含　　义
*	匹配零个或多个字符
?	精确匹配一个字符
[abc]	匹配字符集中的一个字符：a，b 或 c
[a-z]	匹配 a 至 z 范围内的任一个字符
{a,ile,ax}	匹配一个字符或一个字符集
~	代替用户的工作目录
\	转义或禁用元字符

shell 通过对它的元字符求值，将其替换为适当的字母或数字来执行文件名替换。

9.8.1　星号

星号匹配文件名中的零个或多个字符。

范例 9-28

```
1  % ls
   a.c b.c abc ab3 file1 file2 file3 file4 file5
2  % echo *
   a.c b.c abc ab3 file1 file2 file3 file4 file5
3  % ls  *.c
   a.c b.c
4  % rm  z*p
   No match.
```

说明

1. 列出当前目录下的所有文件。

2. echo 程序将它的所有参数打印到屏幕上。星号(也称 splat)是一个通配符，用于匹配文件名中的零个或多个任意字符。当前目录下的所有文件都被匹配并回显到屏幕上。

3. 列出以.c 结尾的文件名。

4. 当前目录下没有以 z 开头的文件名，所以 shell 报告 No match(无匹配)。

9.8.2　问号

问号仅匹配文件名中的一个字符。

范例 9-29

```
1  % ls
   a.c b.c abc ab3 file1 file2 file3 file4 file5
2  % ls ???
   abc ab3
3  % echo How are you?
   No match.
```



```
4   % echo How are you\?
    How are you?
```

说明

1. 列出当前目录下的所有文件。
2. 问号用于匹配文件名中单个字符。所以列出的是所有名字为 3 个字符的文件。
3. shell 查找名字为 you 后跟一个任意字符的文件。当前目录下没有匹配这些字符的文件，因此，shell 打印 No match。
4. 问号前的反斜杠用于关闭问号的特殊含义。此时，shell 把问号当成一个普通的字面字符。

9.8.3 方括号

方括号在文件名中匹配在某组字符或某字符范围内的一个字符。

范例 9-30

```
1   % ls
    a.c b.c abc ab3 file1 file2 file3 file4 file5 file10 file11 file12
2   % ls file[123]
    file1 file2 file3
3   % ls [A-Za-z][a-z][1-5]
    ab3
4   % ls file1[0-2]
    file10 file11 file12
```

说明

1. 列出当前目录下的所有文件。
2. 匹配并列出所有以 file 开头，后跟 1、2 或 3 中任一数字的文件名。
3. 匹配并列出所有以字母(大小写均可)开头，后跟一个小写字母，再跟 1~5 之间的一个数字的文件名。
4. 列出以 file1 开头，后跟 0、1 或 2 中任一数字的文件名。

9.8.4 花括号

花括号({})匹配文件名中的一个字符或字符串。

范例 9-31

```
1   % ls
    a.c b.c abc ab3 ab4 ab5 file1 file2 file3 file4 file5 foo faa fumble
2   % ls f{oo,aa,umble}
    foo faa fumble
3   % ls a{.c,c,b[3-5]}
    a.c ab3 ab4 ab5
```

说明

1. 列出当前目录下的所有文件。
2. 列出以 f 开头，后跟字符串 oo、aa 或 umble 的文件名。如果花括号中出现空格，就

会产生报错信息“Missing }”。

3. 列出以字母 a 开头，后跟.c、c 或 b3、b4、b5 的文件名(方括号可用在花括号中)。

9.8.5 转义元字符

反斜杠用于屏蔽某一单个字符的特殊含义。被转义的字符将只代表其本身。

范例 9-32

```
1   % gotta light?
    No match.
2   % gotta light\?
    gotta: Command not found.
```

说明

1. 这是跟 UNIX 开的一个小玩笑(注：match 译为火柴)。问号是一个用于文件名替换的元字符，其值为某一单字符。shell 在当前工作目录下查找包含字符 l-i-g-h-t，后面还跟了一个字符的文件名。如果找不到满足条件的文件，shell 就会报告 No match。这个例子在一定程度上说明了 shell 解析命令行的顺序，shell 先对元字符求值，然后才设法定位 gotta 命令。

2. 反斜杠保护元字符，使之不被解释，这一行为通常被称作转义元字符。此时 shell 不再显示 No match，而是在搜索路径中查找 gotta 命令，结果显示没有找到。

9.8.6 ~号扩展

对单独的字符~展开的结果是当前用户的主目录的完整路径名。如果把~号写在某个用户名前面，它展开的结果就是该用户的主目录的完整路径名。如果写在路径前面，~号就会展开为用户的主目录，后面跟上路径的剩余部分。

范例 9-33

```
1   % echo ~
    /home/jody/ellie
2   % cd ~/desktop/perlstuff
    % pwd
    /home/jody/ellie/desktop/perlstuff
3   % cd ~joe
    % pwd
   /home/bambi/joe
```

说明

1. ~号扩展为当前用户的主目录。

2. ~号后面跟了一个路径名，它展开的结果是当前用户的主目录，后面跟上 /desktop/perlstuff。

3. 后跟用户名的~号展开为该用户的主目录。本例中，当前工作目录转到了用户 joe 的主目录。



9.8.7 文件名补全：变量 filec

处于交互式运行状态时，C/TC shell 为键入文件名和用户名提供了一种快捷方式。内置变量 filec 被设置后，可用于所谓的文件名补全。如果您键入当前目录下某个文件的名字的前几个字符，然后按下 ESC 键，shell 就会补上该文件名的剩余部分，前提是当前目录下没有名称以这组字符开头的其他文件。如果您输入文件名的部分字符后按下 Ctrl+D 组合键，shell 将打印出名字与这些字符匹配的文件的清单。如果存在多个匹配，终端将发出蜂鸣声。如果输入的字符序列前有一个~号，则 shell 会试着把该序列扩展为一个用户名。

范例 9-34

```
1   % set filec
2   % ls
    rum rumple rumplestilsken run2
3   % ls ru[ESC][6]      # terminal beeps
4   % ls rum^D
    rum rumple rumplestilsken
5   % ls rump[ESC]
    rumple
6   % echo ~ell[ESC]
    /home/jody/ellie
```

说明

1. 设置 filec 这个特殊的 C shell 变量后，就能使用文件名补全功能。
2. 列出当前工作目录下的文件。
3. 尝试文件名补全功能。字母 r 和 u 的组合不唯一，shell 不知道该选哪一个，于是终端发出蜂鸣声。
4. 依次键入字母 r、u、m 后，再按下^D(即 Ctrl+D 组合键)，屏幕上就显示出所有以 rum 开头的文件名的清单。
5. 第一个以 rump 开头的文件名被补全并显示。
6. 如果拼写了一半的用户名前面有一个~号，shell 将设法拼全这个用户的名字，并且显示该用户的主目录。

9.8.8 用 noglob 关闭元字符

一旦设置了变量 noglob，文件名替换功能就被关闭，这意味着所有的元字符都将只代表其自身，它们不再被用作通配符。使用 grep、sed 或 awk 之类的程序搜索模式时，要用到关闭元字符的功能，因为 shell 可能会试着去展开这些程序所使用的元字符。

范例 9-35

```
1   % set noglob
2   % echo * ?? [] ~
    * ?? [] ~
```

⑥ [ESC]代表 ESC 键。

说明

1. 设置变量 noglob。该操作关闭了通配符的特殊含义。
2. 元字符只代表其本身，没有任何其他含义。

9.9 重定向与管道

通常情况下，命令的标准输出(stdout)被发往屏幕，标准输入(stdin)来自键盘，报错信息(stderr)也发往屏幕。shell 允许使用特殊的重定向元字符将输出重定向到某个文件，或将输入重定向为某个文件。重定向操作符(<、>、>>、>&)后面要跟一个文件名，shell 会在执行左端的命令之前打开这个文件。

管道用一个竖杠(|)表示，它允许某个命令的输出作为另一个命令的输入。管道左边的命令执行写操作，因为它要往管道写入内容。管道右边的命令则执行读操作，因为它要从管道读取内容。请参见表 9-5 中列出的重定向和管道元字符。

表 9-5 重定向元字符

<table>
<tr><th>元 字 符</th><th>含 义</th></tr>
<tr><td>command < file</td><td>把命令 command 的输入重定向为文件 file</td></tr>
<tr><td>command > file</td><td>将 command 的输出重定向到 file</td></tr>
<tr><td>command >& file</td><td>将 command 的输出和报错信息重定向到 file</td></tr>
<tr><td>command >> file</td><td>重定向 command 的输出，将其追加到 file 末尾</td></tr>
<tr><td>command >>& file</td><td>重定向 command 的输出和报错信息，将其追加到 file 末尾</td></tr>
<tr><td>command << WORD</td><td>将 command 的输入重定向为从第一个 WORD 处开始，到下一个 WORD 处之间的内容。即 here 文档的开头</td></tr>
<tr><td><input>
WORD</td><td><input>为用户的输入，这些输入被当成用双引号括起来的文本串
WORD 标志 command 的输入到此结束，即 here 文档的结尾</td></tr>
<tr><td>command | command</td><td>用管道将第一个 command 的输出作为第二个文件的输入</td></tr>
<tr><td>command |& command</td><td>用管道将第一个 command 的输出和报错信息作为第二个文件的输入</td></tr>
<tr><td>command >! file</td><td>如果设置了 noclobber 变量，屏蔽其对该命令的影响，打开或重写文件 file</td></tr>
<tr><td>command >>! file</td><td>忽略 noclobber 变量；如果文件 file 不存在，就创建它并将 command 的输出追加到 file 尾部</td></tr>
<tr><td>command >>&! file</td><td>忽略 noclobber 变量；如果文件 file 不存在，就创建它并将 command 的输出和报错信息一起追加到 file 尾部</td></tr>
</table>

9.9.1 重定向输入

输入不一定非得来自键盘，可以将输入重定向为来自某个文件。shell 将打开符号<右边的文件，其左边的程序将从该文件读取输入。如果指定的文件不存在，C shell 将报告错误：No such file or directory。



格式

命令 < 文件

范例 9-36

```
mail bob < memo
```

说明

shell 打开 memo 文件，mail 程序的输入被重定向为该文件。简单说来就是将名为 memo 的文件用 mail 程序发送给用户 bob。

9.9.2 here 文档

here 文档是把输入重定向到命令的另一种方法。它被 shell 脚本用来生成菜单和处理来自其他程序的输入。通常，从键盘接收输入的程序以用户按下 Ctrl+D 组合键作为输入终止的信号。here 文档提供了一种替代的办法来向程序发送输入，而且无需要键入^D 就能结束输入。符号<<后面跟的是一个用户自定义的词，这个词通常被称为终结符(terminator)。输入被导向符号<<左边的命令，并以用户定义的那个终结符为结尾。最后那个终结符必须独占一行，且前后不能有空格。shell 会在 here 文档中执行变量替换和命令替换(here 文档通常出现在 shell 脚本中，用于生成菜单或为 mail、bc、ex、ftp 之类的程序提供输入)。

格式

```
命令 << 标志
  ……输入……
标志
```

范例 9-37

```
(不使用here 文档)
(命令行)
1    % cat
2    Hello There.
     How are you?
     I'm tired of this.
3    ^D

(输出)
4    Hello There.
     How are you?
     I'm tired of this.
```

说明

1. cat 程序没有参数，它等待来自键盘的输入。
2. 用户用键盘进行输入。
3. 用户键入^D 以结束对 cat 程序的输入。
4. cat 程序把输出结果发送到屏幕上。

范例 9-38

```
(使用 here 文档)
(命令行)
1    % cat << DONE
2    Hello There.
     How are you?
     I'm tired of this.
3    DONE

(输出)
4    Hello There.
     How are you?
     I'm tired of this.
```

说明

1. cat 程序接收从第一个 DONE 到结尾的 DONE 之间的内容作为输入。DONE 是用户定义的终结符。

2. 这几行就是输入。遇到单词 DONE 后，程序不再接收输入。

3. 末尾的终结符标志输入的结束。这个词前后都不能有空格。

4. 首尾两个 DONE 之间的文本就是 cat 命令的输出(从“here”到“here”)，被发送到屏幕上。最后那个 DONE 必须紧挨着左端，前面不能有空格，DONE 的右边也不能有任何文本。

范例 9-39

```
(命令行)
1   % set name = steve
2   % mail $name << EOF
3   Hello there, $name
4   The hour is now `date +%H`
5   EOF
```

说明

1. shell 变量 name 被赋值为用户名 steve(通常，此例会被包含在某个 shell 脚本中)。

2. 变量 name 在 here 文档中被展开。

3. mail 程序开始接受输入，直至遇到终结符 EOF 为止。

4. shell 在 here 文档中执行命令替换，即执行反引号中的命令，将命令的输出替换到字符串中。

5. 遇到终结符 EOF，mail 程序的输入到此结束。

9.9.3 重定向输出

默认情况下，命令或命令组的标准输出通常是发往终端的屏幕。要将标准输出从屏幕重定向到某个文件，就要使用符号>。命令写在符号>的左边，而文件名则写在右边。shell 将打开符号>右边的那个文件。如果这个文件不存在，shell 就会创建它。如果这个文件存在，shell 就会打开并清空它。使用重定向时常常会因疏忽而删除文件(C/TC shell 有一个专



用的变量叫做 noclobber，设置这个变量能够防止重定向破坏已有的文件。请参见 9.9.7 节中的表 9-6)。

格式

命令 > 文件

范例 9-40

```
cat file1 file2 > file3
```

说明

将文件 file1 和 file2 的内容串接起来，并且将输出结果发送到文件 file3。记住 shell 会在执行 cat 命令之前打开文件 file3。如果 file3 已经存在并且包含数据，它的数据将丢失。如果 file3 尚不存在，shell 会创建它。

9.9.4 将输出追加到已有文件

要将输出追加到某个已有文件中，应该使用符号>>。如果符号>>右边的文件不存在，shell 就会创建它。如果文件已存在，shell 就打开该文件，并将输出追加到它的末尾。

格式

命令 >> 文件

范例 9-41

```
date >> outfile
```

说明

重定向 date 命令的标准输出，将其追加到文件 outfile 中。

9.9.5 重定向输出和报错信息

符号>&用于将标准输出和标准错误输出一起重定向到某个文件。通常，命令执行的结果要么是成功并将输出发送到 stdout，要么是失败并将报错信息发往 stderr。有些递归执行的程序(比如 find 和 du)在目录树中移动时，会将标准输出和标准错误输出一同发往屏幕。如果使用符号>&，就能将标准输出和标准错误输出全都保存在某个文件中以便查看。C/TC shell 没有提供仅重定向标准错误输出的符号，但是可以通过在子 shell 中执行命令来提取标准错误输出。参见范例 9-42 和图 9-2。

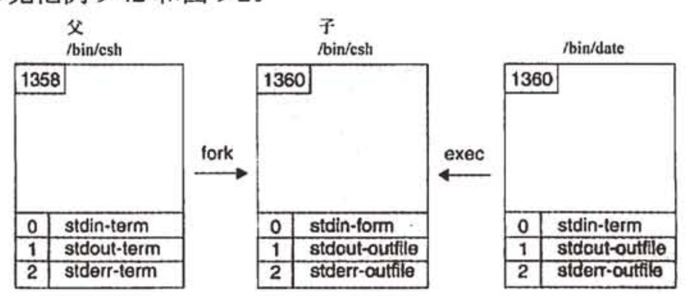


图 9-2 重定向 stdout 和 stderr(请参照范例 9-42)

范例 9-42

```
1   % date
    Tue Aug 9 10:31:56  PDT  2004
2   % date >& outfile
3   % cat outfile
    Tue Aug 9 10:31:56  PDT  2004
```

说明

1. date 命令将输出发往标准输出，即屏幕。

2. 将标准输出和标准错误输出发送到文件 outfile 中。

3. 由于没有错误输出，所以只有标准输出被发送到 outfile 文件中，文件的内容被显示在屏幕上。

范例 9-43

```
1   % cp file1 file2
2   % cp file1
     Usage: cp [-ip] f1 f2; or: cp [-ipr] f1 ... fn d2
3   % cp file1 >& errorfile
4   % cat errorfile
     Usage: cp [-ip] f1 f2; or: cp [-ipr] f1 ... fn d2
```

说明

1. 复制文件时，cp 命令需要知道源文件和目标文件。cp 命令制作了一份 file1 的副本，并将这份副本放到文件 file2 中。由于 cp 命令使用了正确的语法，所以屏幕上没有任何显示，表明文件复制成功了。

2. 这一次漏写了目标文件，cp 命令执行失败，报错信息被发送到 stderr，即终端上。

3. 用符号>&将 stdout 和 stderr 一同发给文件 errorfile。报错信息是这条命令唯一的输出，因此只有这条信息被保存到文件中。

4. 显示文件 errorfile 的内容，其中包含了 cp 命令产生的报错信息。

9.9.6 分离输出与报错信息

用圆括号将命令括起来就能将标准输出和标准错误输出分开。如果将命令写在圆括号中，C/TC shell 就会启动一个子 shell，在这个子 shell 中处理重定向，然后才执行命令。使用范例 9-44 中的方法就能将标准输出与标准错误输出分开。

范例 9-44

(命令行)

```
1   % find . -name '*.c' -print >& outputfile
2   % (find . -name '*.c' -print > goodstuff) >& badstuff
```

说明

1. find 命令以当前目录为起点，查找所有名字以.c 结尾的文件，并将输出结果打印到文件 outputfile 中。如果命令执行过程中发生了错误，报错信息也会写到文件 outputfile 中。

2. find 命令被括在圆括号中。shell 将创建一个子 shell 处理这个命令。创建子 shell 之



前，shell 会先处理括号外的词。也就是说，shell 将打开文件 badstuff 以保存标准输出和标准错误输出。子 shell 启动后，将从父 shell 处继承标准输入、标准输出和标准错误输出。此时子 shell 的标准输入来自键盘，标准输出和标准错误输出都发往文件 badstuff。接下来子 shell 就要处理>操作符，将 stdout 指定到文件 goodstuff。最后，输出将写入文件 goodstuff，报错信息则发给文件 badstuff。请参见图 9-3。

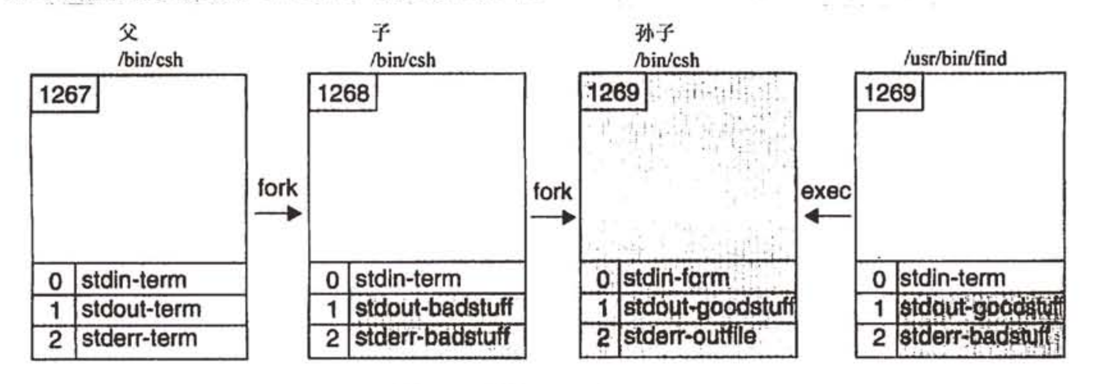


图 9-3 分离 stdout 和 stderr

9.9.7 变量 noclobber

C shell 有一个专用的内置变量叫做 noclobber，可以通过设置该变量来避免在重定向时破坏文件的数据。请参见表 9-6。

表 9-6 noclobber 变量与重定向

	文 件 存 在	文件不存在
未设置 noclobber		
command > file	文件被重写	创建文件
command >> file	向文件追加内容	创建文件
设置了 noclobber		
command > file	发出报错信息	创建文件
command >> file	向文件追加内容	发出报错信息
忽略 noclobber		
command >! file	如果设置了 noclobber 变量，则忽略其对命令 command 的影响，打开或清空文件，将 command 的输出重定向到文件 file	
command >>! file	忽略 noclobber 变量。如果文件 file 不存在，则创建该文件，并将 command 的输出追加到其尾部(参见范例 9-45)	

范例 9-45

```
1   % cat filex
    abc
    123
2   % date > filex
3   % cat filex
```

```
    Wed Aug 5 11:51:04  PDT 2004
4   % set noclobber
5   % date > filex
    filex: File exists.
6   % ls >! filex     # Override noclobber for this command only
    % cat filex
    abc
    ab1
    dir
    filex
    plan.c
7   % ls > filex
    filex:  File exists.
8   % unset noclobber    # Turn off noclobber permanently
```

说明

1. 把文件 filex 的内容显示在屏幕上。
2. 将 date 命令的输出重定向到文件 filex。该文件被清空，其内容被重写。
3. 显示文件 filex 的内容。
4. 设置 noclobber 变量。
5. 因为 filex 已经存在，而且设置了 noclobber 变量，所以 shell 报告文件已存在，不允许将其重写。
6. ls 的输出被重定向到文件 filex，因为操作符>!使 noclobber 的作用被忽略。
7. 符号>!的效果是暂时的。它没有关闭 noclobber，只能在它被使用的地方忽略 noclobber 对命令的影响。
8. 将 noclobber 变量复位。

9.10 变量

C/TC shell 的变量只能保存字符串或字符串集。有些变量被内置在 shell 中，可以通过打开或关闭来对它们进行设置，例如 noclobber 和 filec 这两个变量。其他的变量则被赋给一个串值，例如变量 path。您可以创建自己的变量，用字符串或命令的输出对其赋值。变量的名字要区分大小写，最多可以包含 20 个字符，可用的字符包括字母、数字和下划线。

变量分为两种类型：局部变量和环境变量。局部变量使用 set 命令创建，全局变量使用 setenv 命令创建。变量的作用域就是其可见范围。局部变量只在定义它的 shell 中可见。环境变量的作用域常常被描述为全局的(global)，它们的作用域是当前 shell 和该 shell 派生(启动)的所有进程。

美元符($)是一个特殊的元字符，如果把它写在某个变量名的前面，它就会指示 shell 取出该变量的值。如果给 echo 命令一个变量作为它的参数，echo 命令会在 shell 处理完命令行，执行过变量替换后，显示出该变量的值。

如果把特殊记号$?放在一个变量名的前面，就会显示该变量是否已被设置。如果返回



值是 1，则表示结果为真，即变量已被设置。如果返回的是 0，则表示结果为假，变量尚未被设置。

范例 9-46

```
1  % set filec
2  % set history = 50
3  % set name = George
4  % set machine = `uname -n`
5  % echo  $?machine
   1
6  % echo  $?blah
   0
```

说明

1. 设置内置变量 filec，以激活文件名补全功能。该变量可以打开或关闭。

2. 内置变量 history 的值设为 50，以控制要显示的事件的数量。

3. 将用户自定义变量 name 的值设为 George。

4. 将用户自定义变量 machine 的值设为这个 UNIX 命令的输出。该命令写在反引号中，这样就能通知 shell 执行命令替换。

5. 在变量名前面加上$?以测试该变量是否已被设置。测试结果是 1(真)，说明该变量已被设置。

6. $?的结果为 0(假)。则变量 blah 尚未被设置。

9.10.1 花括号

花括号用于分隔变量与跟在变量后面的字符。

范例 9-47

```
1  % set var = net
   % echo $var
   net
2  % echo $varwork
   varwork: Undefined variable.
3  % echo ${var}work
   network
```

说明

1. 花括号用于将变量与它后面的字符隔开，此处仅存在变量名，故省略。

2. 未使用花括号时，将 varwork 看成是一个变量。此变量尚未定义，因此 shell 输出报错信息。

3. 使用了花括号后，变量与附在变量后面的字符隔开。$var 将被展开，字符串 work 被加在展开结果的后面。

9.10.2 局部变量

局部变量只能在创建它们的 shell 中被识别。如果在.cshrc 文件中设置某个局部变量，

则每启动一个新 shell，该变量都会被重置。通常，局部变量以小写字符命名。

设置局部变量 如果用来给变量赋值的字符串包含一个以上的单词，就必须为它加上引号。否则，只有第一个词会赋给变量。等号两侧有空格没关系，但是，如果等号的一侧有空格，另一侧也必须有空格。

范例 9-48

```
1   % set round = world
2   % set name = "Santa Claus"
3   % echo $round
     world
4   % echo $name
    Santa Claus
5   % csh               # Start a subshell
6   % echo $name
    name: Undefined variable.
```

说明

1. 局部变量 round 被赋值为 world。

2. 局部变量 name 被赋值为 Santa Claus。双引号阻止了 shell 对 Santa 和 Claus 之间的空格求值。

3. 变量前面的美元符让 shell 执行变量替换，即提取出保存在变量中的值。

4. 执行变量替换。

5. 启动一个新的 C shell 进程(即子 shell)。

6. 在子 shell 中，变量 name 尚未定义。它在父 shell 中被定义为局部变量。

set 命令 set 命令打印出为当前 shell 设置的所有局部变量。

范例 9-49

```
(命令行--Linux/tcsh)
> set
addsuffix
argv          ()
cwd          /home/jody/meta
dirstack      /home/ellie/meta
echo_style both
edit
gid       501
group     ellie
history   500
home      /home/ellie
i        /etc/profile.d/mc.csh
owd      /home/ellie
noclobber
path  (/usr/sbin /sbin /usr/local/bin /bin  /usr/bin /usr/X11R6/bin )
prompt    [%n@%m  %c]#
prompt2   %R?
prompt3   CORRECT>%R  (y|n|e|a)?
savedirs
```




```
shell    /bin/tcsh
shlvl    2
status   0
tcsh     6.07.09
term     xterm
user     ellie
version tcsh 6.07.09 (Astron) 2004-07-07 (i386-intel-linux)
options 8b,nls,dl,al,rh,color
```

说明

为当前 shell 设置的所有局部变量都被打印出来。其中，大多数的变量例如 history、dirstack 和 noclobber 是在.tcshrc 文件中设置的。其他变量，如 argv、cwd、shell、term、user、version 和 status 是预设的内置变量。

范例 9-50

```
(命令行--UNIX/csh)
% set
argv          ()
cwd           /home/jody/ellie
fignore       .o
filec
history       500
home          /home/jody/ellie
hostname      jody
ignoreeof
noclobber
notify
path          (/home/jody/ellie /bin /usr/local /usr/usr/bin/usr/etc .)
prompt        jody%
shell         /bin/csh
status        0
term          sun-cmd
user          ellie
```

说明

为当前 shell 设置的所有局部变量都被打印出来。这些变量大多是在.cshrc 文件中设置的。变量 argv、cwd、shell、term、user 和 status 是预设的内置变量。

只读变量(tcsh) 只读变量是一种局部变量，一旦被设置，不能再更改或重置，否则将产生错误信息(例如在 C Shell 中，将显示“Set: Syntax error”)。所以环境变量不能设为只读。

范例 9-51

```
1   > set -r name = Tommy
2   > unset name
    unset: $name is read-only.
3   > set name = Danny
    set: $name is read-only
```

内置的局部变量 shell 有很多预先定义的变量，这些变量都有自己的定义。其中一部分变量只有开或关这两种状态。例如，如果您设置了 noclobber 变量，该变量就是打开和有

效的。如果将 noclobber 复位，它就被关闭。有些变量在设置时需要有个定义。如果要让内置变量在不同的 C/TC shell 中都有效，通常应该在文件中设置它们。对 C shell 这个文件是.cshrc，对 TC shell 这个文件是.tschrc。前面已经讨论过一些内置变量，包括 noclobber、cdpath、history、filec 和 noglob。内置命令的完整清单请参见 9.20.1 节中的表 9-26。

9.10.3 环境变量

环境变量经常被称作全局变量。环境变量在创建它们的 shell 中定义，并被该 shell 派生的所有 shell 继承。尽管环境变量会被子 shell 继承，但子 shell 中定义的环境变量却不会传回给父 shell。继承是以父进程到子进程的方向进行，而不可能反过来(这点跟实际生活中一样)。通常，环境变量用大写字母命名。

范例 9-52

(命令行)

```
1   % setenv TERM wyse
2   % setenv PERSON "Joe Jr."
3   % echo $TERM
    wyse
4   % echo $PERSON
    Joe Jr.
5   % echo $$         # $$ evaluates to the PID of the current shell
    206
6   % csh             # Start a subshell
7   % echo $$
    211
8   % echo $PERSON
    Joe Jr.
9   % setenv PERSON "Nelly Nerd"
10、% echo $PERSON
    % Nelly Nerd
11  % exit            # Exit the subshell
12  % echo $$
    206
13  % echo $PERSON  # Back in parent shell
    Joe Jr.
```

说明

1. shell 的环境变量 TERM 被设置为 wsye 终端。
2. 用户自定义变量 PERSON 被设置为 Joe Jr.。使用双引号是由于存在空格。
3. 变量名前面的美元符号($)可以使 shell 求出变量的内容，即变量替换。
4. 打印环境变量 PERSON 的值。
5. 变量$$保存当前 shell 的 PID。例子中这个 shell 的 PID 是 206。
6. csh 命令启动一个新的 C shell，称为子 shell。
7. 打印当前 shell 的 PID。这是一个新的 C shell，所以有个不同的 PID 号，这个 PID 是 211。
8. 环境变量 PERSON 被新 shell 继承。



9. 重新设置变量 PERSON 的值为 Nelly Nerd。这个变量将被该 shell 派生的所有子 shell 继承。

10. 打印出变量 PERSON 的新值。

11. 退出子 shell。

12. 现在运行的是原来那个 C shell。为了证明这一点，打印出了 PID，其值为 206，跟启动子 shell 前的 PID 一样。

13. 变量 PERSON 保存的是初始的值。

打印环境变量 printenv(BSD 版本)命令和 env(SVR4 版本)命令都能够在 Linux 环境下工作，它们将打印出当前 shell 及其子 shell 中设置的所有环境变量。在 BSD 和 SVR4 版本的 C shell 上，setenv 命令都能打印出变量和它们的值。

范例 9-53

```
(Linux/tcsh 的例子)

> env or printenv or setenv
USERNAME=root
COLORTERM=rxvt-xpm
HISTSIZE=1000
HOSTNAME=homebound
LOGNAME=ellie
HISTFILESIZE=1000
MAIL=/var/spool/mail/ellie
MACHTYPE=i386
COLORFGBG=0;default;15
TERM=xterm
HOSTTYPE=i386-linux
PATH=/usr/sbin:/sbin:/usr/local/bin:/bin:/usr/bin:/usr/
X11R6/bin:/home/ellie/bin;/root/bash-2.03/:/usr/X11R6/bin:/home/
ellie/bin;/root/bash-2.03/:/usr/X11R6/bin
HOME=/root
SHELL=/bin/bash
PS1=[\u@\h \W]\$
USER=ellie
VENDOR=intel
GROUP=ellie
HOSTDISPLAY=homebound:0.0
DISPLAY=:0.0
HOST=homebound
OSTYPE=linux
WINDOWID=37748738
PWD=/home/ellie
SHLVL=6
_=/usr/bin/env
```

说明

环境变量是为当前会话框和从当前 shell 启动的所有进程设置的。使用内置命令 env 或 printenv 都可以显示环境变量。很多应用程序都需要设置环境变量。例如，mail 命令将变

量 MAIL 设置为用户邮件伪脱机程序的位置，环境变量 DISPLAY 则决定 xterm 程序使用哪个位图显示终端。执行这类程序时，shell 会将环境变量中的信息传给它们。

范例 9-54

```
(UNIX/Solaris/csh 例子)

% env
FONTPATH=/usr/local/OW3/lib/fonts
HELPPATH=/usr/local/OW3/lib/locale:/usr/local/OW3/lib/help
HOME=/home/jody/ellie
LD_LIBRARY_PATH=/usr/local/OW3/lib
LOGNAME=ellie
MANPATH=/ur/local/man:/usr/local/man:/usr/local/doctools/man:/usr/man
NOSUNVIEW=0
OPENWINHOME=/usr/local/OW3
PATH=/bin:/usr/local:/usr:/usr/bin:/usr/etc:/home/5bin:/usr/
  doctools:/usr:.
PWD=/home/jody/ellie
SHELL=/bin/csh
TERM=sun-cmd
USER=ellie
WINDOW_PARENT=/dev/win0
WINDOW_TTYPARMS=
WMGR_ENV_PLACEHOLDER=/dev/win3
```

9.10.4 数组

C shell 的数组很简单，就是括号中用空格或制表符分隔的一列词。数组元素用从 1 开始的下标编号。如有指定的某个下标没有对应的数组元素，shell 就会显示消息“Subscript out of range”（下标越界）。命令替换也会创建数组。如果把记号$#写在数组的名字前面，就会显示出数组中元素的个数。

范例 9-55

```
1  % set fruit = ( apples pears peaches plums )
2  % echo $fruit
   apples pears peaches plums
3  % echo $fruit[1]        # Subscripts start at 1
   apples
4  % echo $fruit[2-4]      # Prints the 2nd, 3nd, and 4th elements
   pears peaches plums
5  $ echo $fruit[6]
   Subscript out of range.
6  % echo $fruit[*]        # Prints all elements of the array
   apples pears peaches plums
7  % echo $#fruit          # Prints the number of elements
   4
8  % echo $fruit[$#fruit]  # Prints the last element
   plums
```



```
9   % set fruit[2] = bananas   # Ressigns the second element
    % echo $fruit
    apples bananas peaches plums
10  % set path = ( ~ /usr/bin /usr /usr/local/bin . )
    % echo $path
    /home/jody/ellie /usr/bin /usr /usr/local/bin .
11  % echo $path[1]
    /home/jody/ellie
```

说明

1. 词表被括在圆括号中。各个词之间用空白符分隔。这个数组的名称是 fruit。
2. 打印数组 fruit 中的词。
3. 打印 fruit 数组的第 1 个元素。数组的下标从 1 开始。
4. 打印 fruit 数组的第 2、3、4 个元素。可以用短划线来指定范围。
5. fruit 数组没有第 6 个元素。显示下标越界。
6. 打印 fruit 数组的所有元素。
7. 数组名前面的$#用于取得数组中元素的个数。结果显示 fruit 数组有 4 个元素。
8. 下标$#fruit 的值是数组中元素的总个数，因此，如果用这个值作为访问数组的下标，即[$#fruit]，就会打印出数组的最后一个元素。
9. 给数组的第 2 个元素赋个新值。然后打印数组，其中第 2 个元数的值已被替换为 bananas。
10. path 变量是 C shell 中一个专用的目录数组，用于查找命令。创建 path 数组后，就能访问和修改单个路径元素。
11. 打印 path 的第 1 个元素。

shift 命令和数组 如果内置命令 shift 的参数是一个数组，它将数组的第 1 个元素左移。数组的长度也相应地减去 1(如果没有给它指定参数，shift 命令将移走内置数组 argv 的第 1 个元素。参见 10.5 节“命令行参数”)。

范例 9-56

```
1   % set names = ( Mark Tom Liz Dan Jody )
2   % echo $names
    Mark Tom Liz Dan Jody
3   % echo $names[1]
    Mark
4   % shift  names
5   % echo $names
    Tom Liz Dan Jody
6   % echo $names[1]
    Tom
7   % set days = ( Monday Tuesday )
8   % shift days
9   % echo $days
    Tuesday
10  % shift days
11  % echo $days
```

```
12  % shift days
    shift: no more words.
```

说明

1. 数组名叫 names，set 命令将用括号中的词表对其赋值，词表中各个词之间用空白符分隔。

2. 打印这个数组。

3. 打印 names 数组的第 1 个元素。

4. 数组被左移一个元素，移走了 Mark 这个词。

5. 执行 shift 操作后，数组少了一个元素。

6. 执行 shift 操作后，Tom 成了数组的第 1 个元素。

7. 创建一个名为 days 的数组。数组 days 有两个元素，Monday 和 Tuesday。

8. 将数组 days 左移一次。

9. 打印数组 days。Tuesday 是剩下的唯一元素。

10. 再次左移数组 days。数组变空。

11. 数组 days 为空。

12. 这一次，试着移动数组 days 将导致 shell 发出一条报错信息，说明它无法从一个空数组中移出元素。

创建字符串的数组　有时需要将引号中的字符串创建一个数组。这可以通过将字符串变量放在一对括号中来实现。

范例 9-57

```
1   % set name = "Thomas Ben Savage"
    % echo $name[1]
    Thomas Ben Savage
2   % echo $name[2]
    Subscript out of range.
3   % set name = ( $name )
4   % echo $name[1] $name[2] $name[3]
    Thomas Ben Savage
```

说明

1. 变量 name 被赋值为字符串"Thomas Ben Savage"。

2. 如果把 name 作为数组，它只有一个元素，即整个字符串。

3. 把变量放到括号中，从而创建出一个名为 name 的词的数组。

4. 显示这个新数组的 3 个元素。

9.10.5　专用变量

C shell 内置了几个由单个字符构成的变量。字符前的$将允许 shell 对变量进行解释。请参见表 9-7。



表 9-7 变量及其含义

变 量	示 例	含 义
$?var	echo $?name	若变量 var 已被设置，返回 1。否则，返回 0
$#var	echo $#fruit	打印数组的元素个数
$$	echo $$	打印当前 shell 的 PID
$<	set name = $<	接收来自用户的一行输入，到换行符结束

范例 9-58

```
1   % set num
    % echo $?num
    1
2   % echo $path
    /home/jody/ellie   /usr/bin/   usr/local/bin
    % echo $#path
    3
3   % echo $$
    245
    % csh              # Start a subshell
    % echo $$
    248
4   % set name = $<
    Christy Campbell
    % echo $name
    Christy Campbell
```

说明

1. 把变量 num 设为空。如果变量已经被设置(为空或某个值)，则$?的结果就是 1，如果变量未被设置，结果就是 0。

2. 打印 path 变量。这是一个包含 3 个元素的数组。变量前面的$#用于提取并打印数组的元素个数。

3. $$是当前进程的 PID。此时，当前进程就是 C shell。

4. 变量$<接收来自用户的一行输入，输入行止于换行符，但不包括它。该输入行被保存在变量 name 中。第 2 条命令显示变量 name 的值。

路径名变量修饰符 将路径名赋给变量时，可以通过给它添加特殊的 C shell 扩展名来使用路径名变量。路径名分为 4 个部分：头(head)、尾(tail)、根(root)和扩展名(extension)。请参见表 9-8 中列出的关于路径名修饰符及其功能的例子。

表 9-8 路径名修饰符

```
set pn = /home/ellie/prog/check.c
```

修 饰 符	含 义	示 例	结 果
:r	根目录	echo $pn:r	/home/ellie/prog/check
:h	头目录	echo $pn:h	/home/ellie/prog
:t	尾部文件	echo $pn:t	check.c
:e	扩展名	echo $pn:e	c
:g	全部文件	echo $pn:gt	参见范例 9-59

范例 9-59

```
1   % set pathvar = /home/danny/program.c
2   % echo $pathvar:r
    /home/danny/program
3   % echo $pathvar:h
    /home/danny
4   % echo $pathvar:t
    program.c
5   % echo $pathvar:e
    c
6   % set pathvar = ( /home/* )
    echo $pathvar
    /home/jody /home/local /home/lost+found /home/perl /home/tmp
7   % echo $pathvar:gt
    jody  local  lost+found  perl  tmp
```

说明

1. 变量 pathvar 被赋值为/home/danny/program.c。
2. 如果把:r 加在变量后面，显示变量的值时就会把扩展名去掉。
3. 如果把:h 加在变量后面，就只显示路径的头。即路径的最后一个元素被去掉。
4. 如果把:t 加在变量后面，就只显示路径尾部(最后一个元素)。
5. 如果把:e 加在变量后面，就只显示路径的扩展名。
6. 变量的值被设为/home/*。星号被扩展为当前目录下以/home 开头的所有路径名。
7. 如果把:gt 加在变量后面，就只显示每个(全部)路径的文件名。

9.11 命令替换

将 UNIX 命令放在反引号中，就能将它的输出结果赋值给变量或字符串，这一操作称为命令替换(键盘上，反引号的位置通常是在字符~的下面)。用来给变量赋值时，命令的输出被保存为一个词表(参见 9.10.4 节“数组”)，而不是一个字符串，这样，表中每一个词都能被单独访问。要访问表中的某个词，只需在变量名后面加上下标。下标从 1 开始。

范例 9-60

```
1   % echo The name of my machine is `uname -n`.
    The name of my machine is stardust.
2   % echo The present working directory is `pwd`.
    The present working directory is /home/stardust/john.
3   % set d = `date`
    % echo $d
    Sat Jun 20 14:24:21 PDT 2004
4   % echo $d[2] $d[6]
    Jun 2004
5   % set d = "`date`"
    % echo $d[1]
    Sat Jun 20 14:24:21 PDT 2004
```



说明

1. UNIX 命令 uname -n 被括在反引号中。遇到反引号时，shell 将执行其中的命令，即 uname -n，然后将命令的输出结果 stardust 替换到字符串中。当 echo 命令把它的参数打印到标准输出时，机器名将是它的一个参数。

2. shell 执行 UNIX 命令 pwd，并将其输出替换到字符串中命令所在的位置。

3. 将 date 命令的输出赋值给局部变量 d。命令的输出被保存为一个词表(数组)。

4. 打印数组 d 的第 2 个和第 6 个元素。数组下标从 1 开始。

5. 命令输出被引在双引号中，所以它是一个字符串而非词表。

词表与命令替换

把命令括在反引号中作为一个值赋给变量时，所得的值是一个数组(词表)。只要在数组名后加上下标，就能访问数组的每个元素。数组的下标从 1 开始。如果所用的下标大于数组中词的个数，C shell 就会显示“Subscript out of range.”(下标越界)。如果命令的输出不止一行，C shell 会删除各行的换行符，并用空格代替。

范例 9-61

```
1   % set d = `date`
    % echo $d
    Fri Aug 27 14:04:49 PDT 2004
2   % echo $d[1-3]
    Fri Aug 27
3   % echo $d[6]
    2004
4   % echo $d[7]
    Subscript out of range.
5   % echo "The calendar for the month of November is `cal 11 2004`"
    The calendar for month of November is November 2004 S M Tu W
    Th F S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
    22 23 24 25 26 27 28 29 30
```

说明

1. 将 date 命令的输出赋给局部变量 d。命令的输出被保存为一个数组。变量 d 的值被打印。

2. 显示数组 d 的前 3 个元素。

3. 显示数组 d 的第 6 个元素。

4. 数组 d 没有第 7 个元素，所以 shell 报告下标越界。

5. 输出结果超过一行。结果中所有换行符都被替换为空格。这可能不是您希望得到的结果。

范例 9-62

```
1   % set machine = `rusers | awk '/tom/{print $1}'`
2   % echo $machine
    dumbo bambi dolphin
3   % echo $#machine
```

```
        3
4   % echo $machine[$#machine]
    dolphin
5   % echo $machine
    dumbo bambi dolphin
6   % shift $machine
    % echo $machine
    bambi dolphin
7   % echo $machine[1]
    bambi
8   % echo $#machine
    2
```

说明

1. 将 rusers 命令的输出结果通过管道发给 awk。如果在某一行找到正则表达式 tom，awk 就打印该行的第一个字段。这时候，第一个字段是用户 tom 所登录的机器名。

2. 用户 tom 登录了 3 台机器。这些机器的名字被打印出来。

3. 在数组名前面加上$#就能得到数组元素的个数。数组 machine 有 3 个元素。

4. 显示数组 machine 的最后一个元素。用数组的元素个数($#machine)作为下标。

5. 显示 machine 数组。

6. shift 命令将数组左移。数组的第一个元素被丢弃，数组的下标也被重新编排，还是从 1 开始。

7. 显示执行 shift 操作后数组的第一个元素。

8. 执行 shift 操作后，数组的长度减 1。

9.12 引用

C/TC shell 提供了一整套有特殊含义的元字符。实际上，键盘上非字母或数字的字符几乎都在 C shell 中具有某种特殊含义。下面是部分元字符：

```
* ? [ ] $ ~ ! ^ & { } ( ) > < | ; : %
```

反斜杠和引号用于转义元字符，避免其被 shell 解释。反斜杠用于转义单个字符，引号则用于保护字符串。下面是关于使用引用的一些通用规则。

(1) 引号必须成对出现，而且必须在同一行上配对。可以用反斜杠来转义换行符，这样就能在下一行将引号配对。

(2) 单引号可用于保护双引号，双引号也用于保护单引号。

(3) 单引号保护除历史字符(!)之外的所有元字符不被解释。

(4) 双引号保护除历史字符(!)、变量替换字符($)和反引号(用于命令替换)之外的所有元字符，使其不被解释。



9.12.1 反斜杠

反斜杠用于屏蔽对单个字符的解释，在 C shell 中，反斜杠还是能够转义历史字符(感叹号，也称作 bang)的唯一字符。反斜杠常常被用于转义换行符。shell 不解释引号中的反斜杠。

范例 9-63

```
1   % echo Who are you?
    echo: No match.
2   % echo Who are you\?
    Who are you?
3   % echo This is a very,very long line and this is where \
    break the line.
    This is a very, very long line and this is where
    I break the line.
4   % echo "\\abc"
    \\abc
    % echo '\\abc'
    \\abc
    % echo \\abc
    \abc
5   % echo Wow\!
     Wow!
```

说明

1. 问号用于文件名扩展。它匹配单个字符。shell 在当前目录下查找一个名字是 you 加一个字符的文件。当前目录下并没有这样的文件，所以 shell 返回“No match”来说明找不到符合条件的文件。

2. shell 不会去解释问号，因为它已被反斜杠转义。

3. 用反斜杠转义换行符后，就可以在下一行接着输入字符串。

4. 如果把反斜杠括在单引号或双引号中，它会被原样打印出来。如果不在引号中，反斜杠就转义它自己。

5. 感叹号必须用反斜杠转义，否则 C shell 将执行历史替换。

9.12.2 单引号

单引号必须在同一行内成对出现，它可以转义历史字符(!)以外的所有元字符。之所以不能保护历史字符是基于 C shell 处理字符的顺序，历史字符的处理先于引号，而后于反斜杠。

范例 9-64

```
1   % echo 'I need $5.00'
    I need $5.00
2   % echo 'I need $500.00 now\!\!'
    I need $500.00 now!!
3   % echo 'This is going to be a long line so
    Unmatched '.
4   % echo 'This is going to be a long line so \
```

```
    I used the backslash to suppress the newline'
    This is going to be a long line so
    I used the backslash to suppress the newline
```

说明

1. 字符串被括在单引号中。除历史字符(!)外的所有字符都受到保护，不会被 shell 解释。
2. 必须用反斜杠来保护!!，使之不被 shell 解释。
3. 引号必须在同一行上配成对，否则 shell 就会报告“Unmatched.”。
4. 如果要继续编辑该行，就要用反斜杠来转义换行符。引号在下一行的末尾匹配成对。尽管 shell 忽略了换行符，echo 命令却没有。

9.12.3 双引号

双引号必须成对出现，它允许变量替换和命令替换，以及隐藏历史字符(!)以外的所有元字符。双引号中的反斜杠不能转义美元字符。

范例 9-65

```
1   % set name = Bob
    % echo "Hi $name"
    Hi Bob
2   % echo "I don't have time."
    I don't have time.
3   % echo "WOW!"       # Watch the history metacharacter!
    ": Event not found.
4   % echo "Whoopie\!"
    Whoopie!
5   % echo "I need \$5.00"
    I need \.00
```

说明

1. 局部变量 name 被赋值为 Bob。双引号允许美元符用于变量替换。
2. 双引号中单引号受到保护。
3. 单引号和双引号都不能阻止 shell 解释感叹号。内置命令 history 查找最近一条以双引号开头的命令，但未能找到。
4. 用反斜杠来保护感叹号。
5. 用在双引号中时，反斜杠不能转义美元符。

9.12.4 引用的游戏

只要遵守引用规则，就能在同一命令中以各种组合方式来使用双引号和单引号(关于引用的完整讨论参见第 15 章“调试 Shell 脚本”)。

范例 9-66

```
1   % set name = Tom
2   % echo "I can't give $name" ' $5.00\!'
```



```
    I can't give Tom $5.00!
3   % echo She cried, \"Oh help me\!' "', $name.
    She cried, "Oh help me!", Tom.
```

说明

1. 局部变量 name 被赋值为 Tom。

2. 双引号中单词 can't 里的单引号受到了保护。如果把$5.00 放在双引号中，shell 就会对其中的美元符执行变量替换。因此，字符串$5.00 被括在单引号间，这样就能把美元符变成了字面字符。用反斜杠保护感叹号是因为双引号和单引号都不能阻止 shell 对其进行解释。

3. 第一个会话引号被反斜杠保护。感叹号也被反斜杠保护。最后那个会话引号被括在一对单引号间。单引号将保护双引号。

用:q 修饰符引用变量　修饰符:q 用于替代双引号。

范例9-67

```
1   % set name = "Daniel Savage"
2   % grep $name:q database
      same as
3   % grep "$name" database
4   % set food = "apple pie"
5   % set dessert = ( $food "ice cream")
6   % echo $#dessert
    3
7   % echo $dessert[1]
    apple
8   % echo $dessert[2]
    pie
9   % echo $dessert[3]
    ice cream
10  % set dessert = ($food:q "ice cream")
11  % echo $#dessert
    2
12  % echo $dessert[1]
    apple pie
13  % echo $dessert[2]
    ice cream
```

说明

1. 变量 name 被赋值为字符串 Daniel Savage。

2. 如果把:q 加在变量后面，该变量就被引用。这相当于把变量括在双引号中。

3. 变量$name 两端的双引号将允许执行变量替换，并同时保护所有的空白字符。如果没有这对双引号，grep 程序就会分别在名字为 Savage 和 database 的两个文件中查找 Daniel。

4. 变量 food 被赋值为字符串 apple pie。

5. 变量 dessert 被赋值为由 apple pie 和 ice cream 组成的数组(词表)。

6. 数组 dessert 的元素个数为 3。变量 food 展开时，引号被移走。数组 dessert 有 3 个元素：apple、pie 和 ice cream。

7. 打印 dessert 数组的第 1 个元素。如果未被引用，变量将被扩展为独立的词。

8. 打印 dessert 数组的第 2 个元素。

9. ice cream 加了双引号，所以它被当成一个词。

10. dessert 数组被赋值为 apple pie 和 ice cream。:q 可用于引用变量，其用法与双引号相同，所以，$food:q 就等同于"$food"。

11. 数组 dessert 包括两个字符串，即 apple pie 和 ice cream。

12. 打印 dessert 数组的第 1 个元素，即 apple pie。

13. 打印 dessert 数组的第 2 个元素，即 ice cream。

用:x 修饰符引用变量　如果创建了一个数组，其中某个词包含元字符，可以用:x 来阻止 shell 在执行变量替换时解释这些元字符。

范例 9-68

```
1   % set things = "*.c  a??  file[1-5]"
    % echo $#things
    1
2   % set newthings = ( $things )
    set: No match.
3   % set newthings = ( $things:x )
4   % echo $#newthings
    3
5   % echo "$newthings[1] $newthings[2] $newthings[3] "
    *.c  a??  file[1-5]
6   %  grep $newthings[2]:q filex
    The question marks in a?? would be used for filename expansion
    it is not quoted
```

说明

1. 变量 things 被赋值为一个字符串。其中的每个字符串都包含一个通配符。变量 things 的元素个数为 1，即只有一个字符串。

2. C shell 想根据字符串 things 创建一个数组，它试图展开 things 中的通配符以执行文件名替换，结果产生报错信息 No match。

3. 后缀:x 用于制止 shell 展开变量 things 中的通配符。

4. 数组 newthings 包括 3 个元素。

5. 要打印数组 newthings 中的元素，必须对其进行引用，否则，shell 又会试图展开其中的通配符。

6. :q 引用变量，作用就像用双引号括着这些变量一样。grep 程序将打印出文件 filex 中所有包含模式 a??的行。

9.13　交互式 TC shell 的新特性

TC shell 是在原 Berkeley UNIX C shell 的基础上建立的一个公共增强版。如果使用

Linux，你可以用它来代替传统的 C shell。虽然 tcsh 已经集成在大多数 Linux 发行版中，它还是可以移植到许多其他的操作系统中，例如 Solaris、Windows NT、HP-UX、QNX 等。TC shell 中有很多新特性。本章余下的部分涵盖了这些补充的特性，它们大多数用于简化操作，包括命令行编辑、别出心裁的提示、可编程补全(文件名、命令和变量)、拼写校正等。

这个部分仅关注 TC shell 所添加的新特性，前面部分所涵盖的内容对 C shell 和 TC shell 都适用。

9.13.1 tcsh 的版本

要查出当前使用的 tcsh 是哪个版本，在 shell 提示符后键入下面的命令：

```
which tcsh
```

下面的命令将告诉您 tcsh 的安装目录(通常是/bin)，并显示版本信息：

```
/directory_path/tcsh -c 'echo $version'
```

范例 9-69

```
1   which tcsh
    /bin/tcsh
2   /bin/tcsh -c 'echo $version'
    tcsh 6.07.09 (Astron) 1998-07-07 (i386-intel-linux) options
    8b,nls,dl,al,rh,col
```

9.13.2 shell 提示符

TC shell 有 3 类提示符，主提示符是一个>号，次提示符是一个问号后跟诸如 while、foreach 或 if 之类的 tcsh 命令，第 3 种提示符用于拼写更正功能。主提示符是登录之后显示在终端的提示符，它可以被重置。如果直接在主提示符后编写脚本，而这个脚本需要用到一些 tcsh 编程结构，比如条件词句或循环词句，这时次提示符将会出现，从而可以继续写下一行。次提示符在之后的每行都会显示，直至这个结构完全结束。如果拼写更正被打开，则第三种提示符将出现以确认自动拼写更正(参见 9.16 节“TC shell 拼写校正”)。它包含字符串“CORRECT > corrected command (y|n|e|a)?”。通过在提示字符串中增加特定格式的序列可以自定义提示符，参见表 9-9。

表 9-9 提示字符串

字 符 串	描 述
%/	当前工作目录
%~	当前工作目录，~代表用户主目录，其他用户的主目录用~user 表示
%c[[0]n],%.[[0]n]	当前工作目录的尾部，如果给定 n(一个数字)，则是 n 个尾部
%C	与%c 一样，但不使用~替换
%h, %!, !	当前历史事件数
%M	主机名全称
%m	第一个‘.’之前的主机名

(续表)

字 符 串	描 述
%S(%s)	开始(停止)突出模式
%B(%b)	开始(停止)加粗模式
%U(%u)	开始(停止)下划线模式
%t, %@	12 小时制，AM/PM 时间格式
%T	与%t 类似，但采用 24 小时制
%p	12 小时制，AM/PM 时间格式，精确到秒
%P	与%p 类似，但采用 24 小时制
^c	对 c 的分析与 bindkey 中一样
\c	对 c 的分析与 bindkey 中一样
%%	单个%
%n	用户名
%d	“Day”格式的日期
%D	“dd”格式的日期
%w	“Mon”格式的月份
%W	“mm”格式的月份
%y	“yy”格式的年份
%Y	“yyyy”格式的年份
%l	shell 的 tty
%L	清除从提示符后到所显示内容的末尾或行末尾间的内容
%$	扩展紧跟在$之后的 shell 变量或环境变量
%#	对普通用户是>(或 shell 变量 promptchars 的第一个字符)，对超级用户是#(或 shell 变量 promptchars 的第二个字符)
%{string%}	包含字符串作为字面转义序列。只能用来改变终端属性，不能移动光标位置，不可以为提示符最后序列
%?	在出现提示符之前执行的命令的返回值
%R	对 prompt2 而言，它是分析器的状态。对 prompt3 而言，它是更正的字符串。对历史，它是历史字符串

主提示符 当交互运行时，提示符等待用户输入命令并按下回车键。如果不想使用默认的提示符，可以在.tcshrc 文件中进行设置，这个新的提示符将在当前 shell 以及以后启动的所有 TC shell 中使用。如果只希望为当前会话框进行设置，则应该在 shell 提示符中设置。

范例 9-70

```
1  > set prompt = '[ %n@%m %c]# '
2  [ ellie@homebound ~]# cd ..
3  [ ellie@homebound /home]#  cd ..
```



说明

1. 主提示符被设置为用户登录名(%n)，后跟主机名(%m)、一个空格和当前工作目录。整个字符串被括在方括号中，以一个#结尾。

2. 显示了新提示符。提示符中的~代表用户主目录，cd 命令将目录切换为父目录。

3. 新提示符指出了当前工作目录/home。这样用户始终都能知道当前目录是什么。

次提示符 当在提示符后直接联机编写脚本时，就会出现次提示符。次提示符也可以改变。只要使用了 shell 编程结构，后跟一个换行符，则次提示符就会出现并一直持续到该结构正常结束。只有经常练习才能直接在提示符后正确地编写脚本。一旦键入了命令并按下回车，命令就无法挽回了，tcsh 历史机制并不保存在次提示符后键入的命令。

范例 9-71

```
1     > foreach pal (joe tom ann)
2     foreach? echo Hi $pal
3     foreach? end
      Hi joe
      Hi tom
      Hi ann
4     >
```

说明

1. 这是一个联机编写脚本的例子。因为当输入 foreach 循环后 TC shell 期望更多的输入，于是次提示符就出现了。foreach 将循环处理括号中的每一个词。

2. 循环第一次执行，变量 pal 被赋值为 joe。邮件中 memo 的内容将发给 joe。下次执行循环时，pal 将被赋值为 tom，以此类推。

3. end 语句标志着循环的结束。当括号中的所有项都被处理过后，循环结束并显示主提示符。

4. 显示主提示符。

范例 9-72

```
1     > set prompt2='%R %% '
2     > foreach name ( joe tom ann )
3     foreach % echo Hi $name
4     foreach % end
      Hi joe
      Hi tom
      Hi ann
5     >
```

说明

1. 次提示符 prompt2 被重置为格式化字符串。其中%R 是第二行主提示符后输入的条件或循环结构名。两个百分号的计算结果是一个百分号。

2. 开始了 foreach 命令。这是一个必须以关键字 end 结束的循环结构。次提示符将一直显示直到此循环正确结束。

3. 次提示符是 foreach%。

4. 键入关键字 end 后，循环开始执行。
5. 主提示符再次出现，等待用户输入。

9.14 TC shell 命令行

9.14.1 命令行与退出状态

登录后，TC shell 显示主提示符，默认情况下是符号>。Shell 是一个命令解释器。当 shell 交互地运行时，它从终端读入命令并将其分解为单词。命令行由一个或多个以分隔符(空格或制表符)分开的词(token)组成，以换行符(按下回车键产生的)结束。通常，第一个词是命令，后面的词则是命令的选项或参数。命令可以是 Linux 可执行程序如 ls 或 pwd，可以是别名，可以是内置命令如 cd 或 jobs，也可以是 shell 脚本。命令可以包含通配符，shell 在分析命令行时必须解释这种特殊字符。如果命令行的最后一个字符是反斜杠后跟一个换行符，则命令可以在下一行继续输入[7]。

退出状态和 printexitvalue 变量 当命令或程序结束时，它向父进程返回一个退出状态。退出状态是一个 0~255 之间的数，按惯例，如果退出状态为 0，则程序执行成功，如果退出状态非 0，则程序运行出错。如果程序非正常终止，则退出状态的值需加上 0200。内置命令失败时返回退出状态 1，其他情况下则返回 0。

tcsh 状态变量或？变量将被设置为最后所执行命令的退出状态值。程序是否执行成功是由编写它的程序员决定的。通过设置 tcsh 变量 printexitvalue，任何时候只要程序以非 0 值退出，它的状态将自动被打印。

范例 9-73

```
1  > grep "ellie" /etc/passwd
   ellie:GgMyBsSJavd16s:501:40:E Quigley:/home/jody/ellie:/bin/tcsh
2  > echo $status  or echo $?
   0
3  > grep "nicky" /etc/passwd
4  > echo $status
   1
5  > grep "scott" /etc/passsswd
   grep: /etc/passsswd: No such file or directory
6  > echo $status
   2
7  > set printexitvalue
   > grep "XXX" /etc/passwd
   Exit 1
   >
```

说明

1. grep 程序在/etc/passwd 文件中寻找模式“ellie”并匹配成功。/etc/passwd 文件中相应

[7] 命令行的长度至少为 256 个字符。



的行被显示出来。

2. 变量 status 被设置为 grep 命令的退出值 0 以表示执行成功。变量?也用于保存这个退出状态。bash shell 和 ksh shell 使用?变量来检查退出状态(csh 不使用)。

3. grep 程序在/etc/passwd 文件中没有找到用户 nicky。

4. grep 程序不能找到模式，于是返回退出状态 1。

5. 因为不能打开文件/etc/passwd，grep 命令失败。

6. grep 未能找到文件，于是返回退出状态 2。

7. 设置 tcsh 特定变量 printexitvalue。它将自动打印所有返回状态非 0 的命令的退出状态值。

9.14.2 TC shell 命令行历史

TC shell 内置了历史机制。它在内存中保持一个按序编号的命令列表，这些命令是在命令行键入的，称为事件。历史机制不仅保存这些事件，还保存在终端输入这些命令的时间。当 shell 从终端读取命令时，它将命令行分解为单词(使用空格作为单词分隔符)，然后保存至历史列表，解析并执行它。之前键入的命令也总会被保存。任何时候都可以从历史列表中取回某个命令并执行，而无需再次键入命令。登录到会话中时，所键入的命令将会不断地加入到历史列表中去直至退出会话。退出时，历史列表可以被保存在用户主目录中一个名为.history[⑧]的文件中。历史列表和历史文件这两个术语有时会导致混乱。历史列表指的是当前保存在 shell 内存中的命令行列表。历史文件，通常是.history 文本文件，则用来保存那些将来需要使用的命令。内置变量 savelist 用于在注销时将历史列表存入.history 文件中并在启动时将历史文件的内容载入内存(参见表 9-10 中历史命令的-S 选项和-L 选项)。内置命令 history 显示历史列表，它支持多种可以控制列表显示方式的参数。

表 9-10 history 命令及选项

选　项	含　义
-c	清空内存中的历史列表，而非历史文件
-h	打印不带序号的历史列表
-L[filename]	将历史文件(.history 或 filename)追加到历史列表中
-M[filename]	与-L 类似，但它是将历史文件中的内容与历史列表合并而不是追加
n	n 是一个数字，用于控制显示的行数。例如，history 5
-r	逆序打印历史列表
-S[filename]	将历史列表保存至.history 或给定的 filename 中
-T	以说明的形式打印时间戳

尽管历史文件默认的名字是.history，但它可以通过 shell 内置变量 histfile 重新指定。shell 变量 history 用于设置所显示命令的数量，设置 histdup 变量可以确保重名的项不被加入到

⑧ 通过为 shell 变量 histfile 设置新的名字可以改变.history 的文件名。

历史文件中。

范例 9-74

```
(命令行)
> history
1 17:12   cd
2 17:13   ls
3 17:13   more /etc/fstab
4 17:24   /etc/mount
5 17:54   sort index
6 17:56   vi index
```

说明

历史列表显示出最近命令行键入的命令。列表中的每个事件都带有数字前缀(即事件编号)以及事件被键入的时间。

history 变量　TC shell 变量 history 用于设置要从历史列表中显示的事件数。通常情况下，这个变量在用户初始化文件/etc/.cshrc 或~/.tcshrc 中已设置，其默认值为 100。也可以为 history 变量指定候选值以控制历史列表显示的格式。这个值使用与主提示符相同的格式序列(参见表 9-10)。history 默认的字符串格式为%h\t%T\t%R\n。

范例 9-75

```
1 set history=1000
2 set history= ( 1000 '%B%h %R\n' )
3 history
  136 history
  137 set history = ( 1000 '%B%h %R\n' )
  138 history
  139
  140 pwd
  141 cal
  141 pwd
  142 cd
```

说明

1. 使用 history 命令，将终端最近键入的 1000 个命令显示在屏幕上。

2. 显示最近键入的 1000 条命令。格式字符串的设置将使得历史列表以粗体(%B)显示。首先是事件编号(%h)，然后是一个空格，最后是所键入的命令(%R)以及一个换行符(\n)。

3. 如果键入 history 命令，便可以看到新格式。这里只列出了实际历史列表的一部分。

savehist 变量与历史保存　要保存跨登录会话框的历史事件，必须设置 savehist 变量。通常这个变量在用户初始化文件.tcshrc 中设置。如果 savehist 赋的第一个值是数字，且同时还设置了 history 变量，则该数字的值不能超过 history 变量中所设的值。如果第二个值是 merge，则历史列表将与已有的历史文件合并，而并不替换它。历史文件将保存最近的事件，并按时间戳排序。

范例 9-76

```
1 set savehist
```




```
2   set savehist = 1000
3   set savehist = 1000 merge
```

说明

1. 历史列表中的命令被保存在历史文件中，下次登录时它将位于历史列表的起始位置。

2. 历史文件被历史列表中的最近 1000 条命令所替换并保存。下次登录时它将会显示出来。

3. 注销时，当前历史列表将与现有的历史文件合并，而不是取代现有的历史文件。登录后它们将被加载至内存。

显示历史　history 命令用于显示历史列表中的事件。history 命令还带有选项可以控制事件的数量和将要显示的事件的格式。事件的编号不一定要从 1 开始。如果当前历史列表中有 100 条命令，而 history 变量被设置为 25，则只能看到最近保存的 25 条命令。

范例 9-77

```
1   > set history = 10
2   > history
    1 ls
    2 vi file1
    3 df
    4 ps -eaf
    5 history
    6 more /etc/passwd
    7 cd
    8 echo $USER
    9 set
    10 ls
```

说明

1. history 变量被设为 10。这样历史列表中即使有更多命令，也仅显示最近的 10 条命令。

2. 显示历史列表中最近的 10 个事件。每条命令都被编号。

范例 9-78

```
1   > history -h      # print without line numbers
    ls
    vi file1
    df
    ps -eaf
    history
    more /etc/passwd
    cd
    echo $USER
    set
    history -n
2   > history -c
```

说明

1. 带上选项 h，历史列表将不显示行号。
2. 带上选项 c，历史列表将被清空。

范例 9-79

```
> history -r      # print the history list in reverse
11 history -r
10 history -h
 9 set
 8 echo $USER
 7 cd
 6 more /etc/passwd
 5 history
 4 ps -eaf
 3 df
 2 vi file1
 1 ls
```

说明

历史列表以逆序方式显示。

范例 9-80

```
> history 5       # prints the last 5 events on the history list
7   echo $USER
8   cd
9   set
10  history -n
11  history 5
```

说明

显示了历史列表中最近的 5 个事件。

访问历史文件中的命令　有几种方法可以访问和重复历史列表中的命令。tcsh 新增了一项好用的特性：使用方向键访问历史。使用上下方向键在历史列表中滚动，使用左右键在命令行间移动，进行编辑。可以使用历史替换机制重新执行命令或修复拼写错误，也可以使用内置编辑器 emacs 或 vi 得到、编辑并执以前的命令。我们将全面介绍这些方法，从而使得您可以选择适合自己的最佳工作方式。

1. 方向键

访问历史列表中的命令，可以使用键盘上的上下方向键在历史列表中上下移动，也可以使用左右方向键在命令行中左右移动。使用标准键可以对历史列表中的任何行执行删除、退格等操作。一旦编辑好命令行，按下回车键将使得命令行重新被执行。还可以使用标准的 emacs 或 vi 命令来编辑历史列表(请参见 9.14.3 节中的表 9-13 和表 9-14)。方向键在 vi 和 emacs 中指定的行为是相同的(参见表 9-11)。

表 9-11 方向键用法

方 向 键	功 能
↑	向历史列表上方移动
↓	向历史列表下方移动
→	将光标向历史命令右方移动
←	将光标向历史命令左方移动

2. 重新执行与!!命令

要重新执行历史列表中的命令，可以使用感叹号(bang)开始历史替换。感叹号可以从行的任意位置开始并可以被反斜杠转义。如果！后面跟的是空格、制表符(tab)或者是换行符，它将不被解释。有很多种方法使用历史替换来指定历史列表的哪个部分需要重新执行(参见后面的表 9-12)。如果键入两个感叹号(!!)，则最后一个命令将被重新执行。如果键入一个感叹号后跟一个数字，则历史列表中该数字对应的命令将被执行。如果键入一个感叹号和一个字母，则最近的以该字母开头的命令将被执行。^也被用来作为编辑前一个命令的快捷方式。

执行完历史替换后，历史列表将更新为命令中替换生成的结果。例如，如果键入!!，则前一个命令被重新执行并以扩展形式保存在历史列表中。如果希望这个命令以字面的形式加入到历史列表中，则需要设置 shell 变量 histlit。

范例 9-81

```
1   > date
    Mon Feb  8 12:27:35 PST 2004
2   > !!
    date
    Mon Aug  10 12:28:25 PST 2004
3   > !3
    date
    Mon Aug  10 12:29:26 PST 2004
4   > !d
    date
    Mon Aug  10 12:30:09 PST 2004
5   > dare
    dare: Command not found.
6   > ^r^t
    date
    Mon Apr  10 16:15:25 PDT 2004
7   > history
    1 16:16  ls
    2 16:16  date
    3 16:17  date
    4 16:18  date
    5 16:18  dare
    6 16:18  date
8   > set histlit
```

```
9   > history
      1 16:18  ls
      2 16:19  date
      3 16:19  !!
      4 16:20  !3
      5 16:21  dare
      6 16:21  ^r^t
```

说明

1. 命令行中执行了 date 命令，历史列表被更新。这是列表中最近的命令
2. !!(bang bang)从历史列表中取出最近的命令，这个命令被重新执行。
3. 历史列表中的第 3 个命令被重新执行。
4. 历史列表中以字母 d 开头的最近的命令被重新执行。
5. 键入了错误的命令。
6. ^用于对历史列表中最近的命令进行字母替换。第一次出现的 r 被替换为 t。
7. 历史替换执行之后，history 命令显示历史列表。
8. 通过设置 histlit，shell 执行历史替换时，将键入的命令以字面形式增加到历史列表中，也就是说，以其键入的形式保存。
9. 设置了 histlit 之后，history 命令的输出显示了在历史替换执行之前，所键入的原始命令字符串(这只是一个演示，历史编号并不一定准确)。

范例 9-82

```
1   % cat file1 file2 file3
       <contents of files displayed here>
    > vi !:1
     vi file1
2   > cat file1 file2 file3
    <contents of file1、file2 and file3 are displayed here>
    > ls !:2
     ls file2
     file2
3   > cat file1 file2 file3
    > ls  !:3
     ls file3
     file3
4   > echo a b c
     a b c
    > echo !$
     echo c
     c
5   > echo a b c
     a b c
    > echo !^
     echo a
     a
6   > echo a b c
     a b c
```



```
    > echo !*
     echo a b c
     a b c
7   > !!:p
    echo a b c
```

说明

1. cat 命令在屏幕上显示文件 file1 的内容。历史列表被更新。命令行被分解为词，第一个词的编号为 0。如果词的编号前面是个冒号，这个词将从历史列表中被取出。符号!:1 的意思是：取出历史列表中最近命令的第 1 个参数，在命令字符串中替换。最近命令的第一个参数是 file1(编号为 0 的词是命令本身)。

2. 最近命令的第 2 个参数 file2 替换了!:2 并作为一个参数传给了 ls。结果打印出 File2(File2 是第 3 个词)。

3. ls !:3 含义为：定位历史列表中的最近命令并取出第 4 个词(词的编号从 0 开始)，将其作为一个参数传给命令 ls(File3 是第 4 个词)。

4. bang(!)加一个美元符号($)引用的是历史列表中最近命令的最后一个参数。最后一个参数是 c。

5. ^代表命令之后的第一个参数。bang(!)加上^引用的是历史列表中最近命令的第一个参数。最近命令的第一个参数是 a。

6. 星号(*)代表命令之后所有的参数。bang(!)加上*引用的是历史列表中最近命令的所有参数。

7. 打印历史列表中的最近一条命令，但不执行。历史列表被更新。现在可以对此行执行^替换。

表 9-12 替换与历史

事件标志符	含 义
!	表示历史替换的开始
!!	重新执行前一条命令
!N	重新执行历史列表中的第 N 条命令
!-N	重新执行历史开表中从当前命令开始向后的第 N 条命令
!string	重新执行以 string 开头的最近一条命令
!?string?	重新执行包含字符串 string 的最近一条命令
!?string?%	重新执行参数包含字符串 string 的最近的命令行
!^	在当前命令行使用最近的 history 命令的第一个参数
!*	在当前命令行使用最近的 history 命令的所有参数
!$	在当前命令行使用最近的 history 命令的最后一个参数
!! string	将字符串追加到前一条命令并执行
!N string	将字符串追加到第 N 条命令并执行
!N:s/old/new/	在向前的第 N 条命令中，用新字符串替换第一次出现的旧字符串
!N:gs/old/new/	在向前的第 N 条命令中，用新字符串替换所有的旧字符串

(续表)

事件标志符	含　义
^old^new^	在前一条历史命令中，用新字符串替换所有的旧字符串
command !N:wn	执行追加一个参数(wn)的当前命令，追加的参数来自向前第 N 条命令，wn 是一个数字(以编号 0 为起始)，用于定位前面命令行中的词，0 代表命令本身，1 代表第一个参数，以此类推
!N:p	将命令放在历史列表的最底端并打印出来，但并不执行

9.14.3　内置命令行编辑器

可以使用与 emacs 或 vi 编辑器相同类型的键序来编辑命令行。也可以使用编辑器命令在历史列表中上下滚动。一旦找到命令，就可以编辑它，并通过按下回车键，可以重新执行它。当编译 shell 时，它为 emacs 器绑定了一个默认的按键集合。

bindkey 内置命令　内置命令 bindkey 用于指定用 vi 或 emacs 来进行命令行编辑，并可以列出和设置各编辑器的按键绑定。使用 vi 作为命令行编辑器，则执行带-v 选项的 bindkey 命令：

```
bindkey -v
```

返回到 emacs，需键入：

```
bindkey -e
```

查看编辑器命令列表及每个命令的简短描述，需键入：

```
bindkey -l
```

查看实际按键及它们的绑定情况，需键入：

```
bindkey
```

要将按键绑定到命令，查看“按键绑定”。

内置编辑器 vi　要编辑历史列表，就转到命令行并按下 Esc 键。然后，按下 K 键可以在历史列表中向上滚动，按下 J 键可以向下滚动，就如同 vi 的移动键。要对某个命令进行编辑，可使用 vi 中的标准键来左右移动、删除、插入以及改变文本(参见表 9-13)。编辑完成之后，按下回车键。命令将被执行并加入到历史列表的底部。如果希望增加或插入文本，那么就使用插入命令(i，e，o，0 等)。请记住，vi 有两种模式：命令模式和插入模式。当键入文本时始终处于插入模式，按下 Esc 键可以返回到命令模式。

表 9-13　vi 命令

命　令	功　能
在历史文件中移动	
Esc K 或+	在历史列表中向上移
Esc J 或-	在历史列表中向下移



(续表)

命　令	功　能
G	移动至历史文件的第一行
5G	移动至历史文件的第 5 条命令
/string	在历史文件中向上搜索字符串
?	在历史文件中向下搜索字符串
在命令行中移动	
h	向行的左方移动
l	向行的右方移动
b	向后移动一个词
e 或 w	向前移动一个词
^或 0	移至行的第一个字符的开始处
$	移至行尾
使用 vi 编辑	
a A	追加文本
i I	插入文本
dd dw x	将文本删除至缓冲(行、词或字符)
cc C	改变文本
u U	撤消
yy Y	复制一行至缓冲区
p P	将复制过的或删除的行放到当前行的下面或上面
r R	替换一行中的某个字符或任意数量的文本

内置编辑器 emacs　如果您使用 emacs 作为内置编辑器，那么与 vi 类似，也要从命令行开始。若按下 Ctrl+P 组合键则会在历史文件中向上滚动，按下 Ctrl+N 组合键则向下滚动。使用 emacs 编辑命令改变或更正文本，然后按下回车键，该命令将被重新执行。参见表 9-14。

表 9-14　emacs 命令

命　令	功　能
Ctrl+P	在历史文件中向上移动
Ctrl+N	在历史文件中向下移动
Esc <	移至历史文件第一行
Esc >	移至历史文件最后一行
Ctrl+B	向后移动一个字符
Ctrl+R	向后搜索字符串
Esc B	向后移动一个词
Ctrl+F	向前移动一个字符
Esc F	向前移动一个词
Ctrl+A	移至行首

(续表)

命　令	功　能
Ctrl+E	移至行尾
Esc <	移至历史文件第一行
Esc >	移至历史文件最后一行
使用 emacs 编辑	
Ctrl+U	删除当前行
Ctrl+Y	将行向后移
Ctrl+K	删除光标至行尾的内容
Ctrl+D	删除一个字母
Esc D	向前删除一个词
Esc H	向后删除一个词
Esc (space)	在当前光标位置设置一个标志
Ctrl+X Ctrl+X	交换光标与标记
Ctrl+P Ctrl+Y	将光标至标记之间的部分移至缓冲(Ctrl+P)并使它后移(Ctrl+Y)

绑定按键　内置命令 bindkey 可列出所有的标准按键绑定，包括 emacs 按键绑定和 vi 按键绑定。按键绑定分成 4 组：标准按键绑定(Standard key bindings)、备用按键绑定(Alternative key bindings)、多字符按键绑定(Multi-character bindings)和方向按键绑定(Arrow key bindings)。bindkey 也可用来改变当前的按键绑定。

范例 9-83

```
1  > bindkey
   Standard key bindings
   "^@"         ->  is undefined
   "^A"         ->  beginning-of-line
   "^B"         ->  backward-char
   "^C"         ->  tty-sigintr
   "^D"         ->  list-of-eof
   "^E"         ->  end-of-line
   "^F"         ->  forward-char
   "^L"         ->  clear-screen
   "^M"         ->  newline
       ...        ...
   Alternative key bindings
   "^@"         -> is undefined
   "^A"         -> beginning-of-line
   "^B"         -> is undefined
   "^C"         -> tty-sigintr
   "^D"         -> list-choices
   "^E"         -> end-of-line
   "^F"         -> is undefined
   ......       ......
```

```
Multi-character bindings
"^[[A"        ->   up-history
"^[[B"        ->   down-history
"^[[C"        ->   forward-char
"^[[D"        ->   backward-char
"^[OA"        ->   up-history
"^[OB"        ->   down-history
      ... ...
Arrow key bindings
down          ->   down-history
up            ->   up-history
left          ->   backward-char
right         ->   forward-char
```

bindkey 命令加上-l 选项将列出编辑器命令及它们的作用。参见范例 9-84。

范例 9-84

```
> bindkey -l

  backward-char
          Move back a character
  backward-delete-char
          Delete the character behind cursor
  backward-delete-word
          Cut from beginning of current word to cursor - save in cut buffer
  backward-kill-line
          Cut from beginning of line to cursor - save in cut buffer
  backward-word
          Move to beginning of current word
  beginning-of-line
          Move to beginning of line
  capitalize-word
          Capitalize the characters from cursor to end of current word
  change-case
          Vi change case of character under cursor and advance one character
  change-till-end-of-line
          Vi change to end of line
  clear-screen
  Standard key bindings
          ..... ...
```

如后面的范例 9-85 所示，bindkey 命令还可以显示单个按键绑定的值。默认显示的是 emacs 键的映射，使用-a 选项的 bindkey 命令可以显示 vi 键的映射。其中，bindkey 的参数可以指定为特定的字符序列，可以用它们来代表按键序列及后面它们所绑定的编辑命令键。表 9-15 是这些按键绑定字符列表。您不仅可以将按键绑定到 emacs 或 vi 编辑器命令，还可以绑定到 Linux 命令以及字符串。

表 9-15　按键绑定字符

字　　符	含　　义
^C	Ctrl+C
^[	Escape
^?	Delete
\a	Ctrl+G(bell)
\b	Ctrl+H(backspace)
\e	Esc(escape)
\f	Formfeed
\n	Newline
\r	Return
\t	Tab
\v	Ctrl+K(vertical tab)
\nnn	ASCII octal number

表 9-16 按键绑定选项

bindkey	列出所有的按键绑定
bindkey -a	允许使用备选的按键映射
bindkey –d	恢复默认绑定
bindkey –e	使用 emacs 绑定
bindkey –l	显示所有的编辑命令及它们的含义
bindkey –u	显示使用信息
bindkey –v	使用 vi 键绑定
bindkey key	显示 key 的绑定
bindkey key command	将 key 绑定到 emacs 或 vi 命令
bindkey –c key command	将 key 绑定到 UNIX/linux 命令
bindkey –s key string	将 key 绑定到字符串
bindkey –r key	删除 key 的绑定

范例 9-85

```
1  >bindkey ^L
   "^L"          ->    clear-screen
2  >bindkey ^C
   "^C"          ->    tty-sigintr
3  >bindkey "j"
   "j"           ->    self-insert-command
4  >bindkey -v
5  >bindkey -a "j"
   "j"           ->    down-history
```

说明

1. 带参数^L 的 bindkey 命令显示出 Ctrl+L 组合键(即^L)与哪条命令绑定。^L 用于清屏。



2. Ctrl+C 组合键(^C)绑定到通常用于终止进程的中断信号上。

3. 小写字母"j"是一个 emacs 自插入(self-insert)命令，它只用于将该字符本身插入到缓冲。

4. 看备用 vi 键绑定之前，首先要使用带-v 选项的 bindkey 命令设置 vi 命令行编辑器，正如本行所显示的一样。

5. 带上-a 选项后，bindkey 命令显示"j"的备用按键绑定映射，也就是用在历史列表中向下滚动的 vi 键。

范例 9-86

```
1  > bindkey "^T" clear-screen    # Create a new keybinding
2  > bindkey "^T"
   "^T"        ->      clear-screen
3  > bindkey -a "^T"               # Alternate keybinding undefined
   "^T"        ->      undefined-key
4  > bindkey -a [Ctrl-v Control t] clear-screen   # Create an alternate keybinding
   Press keys one after the other
5  > bindkey -a [Ctrl-v Control t]
   "^T"        ->      clear-screen
6  > bindkey -s '\ehi' 'Hello to you!\n'     # Bind a key to a string
   > echo [Esc]hi   Press escape followed by 'h' and 'i'
    Hello to you!
   >
7  > bindkey '^[hi'
    '^[hi'    ->      "Hello to you!"
8  > bindkey -r '\[hi'          # Remove keybinding
9  > bindkey '\ehi'
   Unbound extended key "^[hi"
10 > bindkey -c '\ex' 'ls | more'   # Bind a key to a command
```

说明

1. Ctrl+T 组合键绑定清屏命令，这是一个默认的 emacs 键映射。这个按键序列开始时并未设置，因此设置后当在这里按下 Ctrl+T 组合键后，将执行清屏动作。

2. 以按键序列为参数的 bindkey 命令在该键序列有绑定时显示其键映射。

3. 以-a 选项及按键序列为参数，bindkey 显示备用键映射 vi 的值。在本例中，bindkey 命令加上-a 选项及按键序列显示出备用映射(vi)并未对该序列进行绑定。

4. 带上-v 选项后，bindkey 可以将任何按键绑定到备用映射 vi。通过按下 Ctrl+V 后跟 Ctrl+T，按键序列被创建并赋值为 clear-screen。Ctrl+V/Ctrl+T 也可以像上例一样表示为“^T”。

5. bindkey 命令显示了^T 的备用映射及其命令。

6. 以-s 为参数的 bindkey 将一个字面字符串绑定到键序列。在这里，字符串“Hello to you!\n”被绑定到转义序列 hi。通过按下 Esc 键然后加上 h 和 i，就可以将该字符串发送到标准输出上。

7. bindkey 命令显示了转义序列 hi 的绑定。^[是 Esc(escape)的另一种表示形式。

8. 带-r 选项的 bindkey 命令用于删除按键绑定。

9. 因为按键绑定被删除，输出显示该扩展按键序列未被绑定。

10. 带-c 选项的 bindkey 命令将一个按键序列绑定到 Linux 命令。在本例中，按下 Esc 键，后面跟上“x”键，就会将命令 ls 的输出通过管道传送给 more 命令。

9.15 TC shell 命令、文件名与变量补齐

为减少击键次数，tcsh 有一种称为补全的机制允许只键入部分的命令、文件名或变量，然后通过按 Tab 键，将词的其余部分补全。

键入某个命令的前几个字母，然后按下 Tab 键，tcsh 将试着补全这个命令名。如果因为不存在这样的命令从而导致 tcsh 无法补全命令，终端将发出蜂鸣声并且光标将处于命令的结尾处。如果有多个命令以这几个字符开头，可以通过按下 Ctrl+D 组合键，将所有以这些字符开头的命令列出。

文件名和变量的补全与命令补全的工作方式相同。对文件名补全，如果存在多个以相同字符开头的文件，tcsh 将补全匹配最短的名字，当文件名补全到字符不一致的位置时，会出现闪烁的光标请您补全其余的部分。参见范例 9-87。

9.15.1 autolist 变量

如果设置了 autolist 变量并且存在多个可能的补全，当按下了 Tab 键后，无论执行的是命令补全、变量补全还是文件名补全，其对应的所有可能的命令、变量或文件名都会被列出。

范例 9-87

```
1   > ls
    file1 file2 foo foobarckle fumble
2   > ls fu[tab]          # expands to filename to fumble
3   > ls fx[tab]          # terminal beeps, nothing happens
4   > ls fi[tab]          # expands to file_(_ is cursor)
5   > set autolist
6   > ls f[tab]           # lists all possibilities
    file1 file2 foo foobarckle fumble
7   > ls foob[tab]        # expands to foobarckle
8   > da[tab]             # completes the date command
    date
    Fri Aug 9 21:15:38 PDT 2004
9   > ca[tab]             # lists all commands starting with ca
    cal   captoinfo  case   cat
10  > echo $ho[tab]me # expands shell variables
    /home/ellie/
11  > echo $h[tab]
    history home
```

说明

1. 列出当前工作目录中的所有文件。
2. 键入 fu 之后按下 Tab 键，文件名将补全为 fumble 并被列出。
3. 因为没有任何以 fx 开头的文件，所以终端发出蜂鸣声，光标停留在原地并不做任



何补全。

4. 若存在许多以 fi 开头的文件，文件名将试着补全直至遇到不相同的字符为止。如果按下 Ctrl+D 组合键，会显示所有符合当前拼写的文件。

5. 设置了 autolist 变量。如果当前有多种选择，当按下 Tab 键之后，autolist 显示所有的可能。

6. 按下 Tab 键后，打印出以 f 开头的文件列表。

7. 按下 Tab 键后，文件名被扩展为 foobarckle。

8. 在 da 后按下 Tab 键后，唯一的以 da 开头的命令是 date。命令名被扩展并执行。

9. 因为设置了 autolist，当在 ca 后按下 Tab 键后，所有以 ca 开头的命令都被列出。如果没有设置 autolist，则需要按下 Ctrl+D 组合键才能得到列表。

10. 词前面的$意思是指当按下 Tab 键后，shell 应该执行变量扩展以补全该词。变量 home 被补全。

11. 这个例子中的变量补全是不确定的。当按下 Tab 键试着补全该变量时，所有可能的 shell 变量都被列出。

9.15.2 fignore 变量

设置 shell 变量 fignore，可以在使用文件名补全时忽略特定后缀的文件。例如，也许您并不想扩展以.o 结尾的文件，因为它们是不可读的对象文件。或者，您不希望在执行文件名扩展时无意中删除后缀为.gif 的文件。无论出于何种原因，都可以通过将一系列后缀赋给 fignore 变量，从而将使用这些后缀的文件排除在文件名扩展之外。

范例 9-88

```
1   > ls
    baby       box.gif   file2    prog.c
    baby.gif   file1     file3    prog.o
2   > set fignore = (.o .gif )
3   > echo ba[tab]      # Completes baby but ignores baby.gif
    baby
4   > echo box[tab].gif  # fignore is ignored if only one completion is possible
    box.gif
5   > vi prog[tab]          # expands to prog.c
    Starts vi with prog.c as its argument
```

说明

1. 列出当前工作目录中的所有文件。注意，有些文件名含有后缀。

2. 变量 fignore 允许您设置文件名后缀，当执行文件名补全时，含有这些后缀的文件被忽略。所有以.o 或.gif 结尾的文件都将被忽略。

3. 按下 Tab 键，仅列出文件 baby，而没有列出 baby.gif。这是因为.gif 文件被忽略了。

4. 尽管.gif 后缀应该被忽略，但 fignore 在没有其他补全可供选择时将不生效。再比如若第 3 行中不存在不带.gif 后缀的同名文件，则结果显示为 baby.gif。

5. 调用 vi 编辑器时，prog 被扩展为 prog.c。

9.15.3 shell 变量 complete

这个命令可以做很多工作！要想从 tcsh 帮助手册中学习它的所有功能有些麻烦，因此，仅对一些例子进行说明以帮助您快速入门。它可以用来控制补全的类型。例如，也许您只想对目录名进行扩展，或者根据文件在命令行中的位置对其进行扩展，或者希望扩展一些特定的命令并把其他命令排除在外，甚至创建一个可以扩展的词列表。无论你希望用哪种类型补全，设置 shell 的变量 complete 都可以实现。

如果将 complete 变量设置为 enhance，文件名补全甚至可用于更复杂的情况。比如：Tab 补全可以忽略大小写，连字符、句号和下划线可以被当作词分隔符，连字符与下划线可以等价。

范例 9-89

```
1   > set complete=enhance
2   > ls g..[tab]   # expands to gawk-3.0.3
    gawk-3.0.3
3   > ls GAW[tab]   # expands to  gawk-3.0.3
    gawk-3.0.3
```

说明

1. 通过将 shell 变量 complete 设置为 enhance，Tab 补全将忽略大小写，把连字符、句号和下划线当作词分隔符，连字符与下划线视为等价。

2. 在 complete 变量被设置为 enhance 的情况下，文件名补全将 g..扩展为以一个 g 开头，后跟任意两个字符(..)，接着是任意字符(包括连字符、句号等)的文件名。

3. 在 complete 变量被设置为 enhance 的情况下，文件名补全将 GAW 扩展为以 GAW 开头，后接任意字符的文件。其中，GAW 可以是任意的大小写组合，其余字符也包括连字符、句号和下划线。

9.15.4 编程补全

为实现一个特别的功能有时需要灵活定制补全的方式，这时可以对补全进行编程，然后将其存储在~/.tcshrc 文件中，使得每次启动一个新的 TC shell 都可以将这种定制的补全作为 tcsh 环境的一部分。编程补全的目的是提高效率并自动选择受影响的命令和参数的类型(用于词补全的 Tab 键和显示所有可能文件名的 Ctrl+D 组合键仍然适用，与简单补全时的用法相同)。

补全的类型 补全有 3 种类型：p 类型、n 类型和 c 类型。p 类型的补全是位置相关的。它是基于命令行中词的位置来判定补全执行的方式。位置 0 是命令，位置 1 是第一个参数，位置 2 是第二个参数，以此类推。例如，你希望确保每次为内置命令 cd 执行命令补全时，仅当 cd 命令的第一个(仅有的一个)参数为一个目录名，而不是其他内容时才执行补全。那么你可以这样编程：

```
complete  cd   'p/1/d/'
```

complete 命令后面跟着 cd 命令和补全规则。p 代表命令行中词的位置。命令 cd 的位置为 0，它的第一个参数位置为 1。规则的模式部分用斜线围起来(p/1/指的是位置 1，即 cd



命令的第一个参数)，它会受到补全规则的影响。模式的 d 部分称为词类型。表 9-17 是关于词类型的完整列表。词类型 d 的含义为仅有目录会受到补全的影响。如果文件名或别名作为 cd 命令的第一个参数，它将不会被补全。规则指出，无论何时尝试对 cd 命令按下 Tab 键进行补全，也只能在 cd 命令的第一个参数为目录的情况下才能真正补全，而按下 Ctrl+D 组合键仅会在匹配不确定，即存在多个可能的补全时才列出目录。范例 9-90 是一个 p 类型补全的例子。

表 9-17　补全词类型

词	类　型
a	别名
b	编辑器按键绑定命令
c	命令(内置或外部命令)
C	以给定路径前缀开头的外部命令
d	目录
D	以给定路径前缀开头的目录
e	环境变量
f	文件名(非目录)
F	以给定路径前缀开头的文件名
g	组名
j	作业
l	界限
n	空
s	shell 变量
S	信号
t	无格式(“文本”)文件
T	以给定路径前缀开头的无格式(“文本”)文件
v	任意变量
u	用户名
X	补全所定义的命令名
x	与 n 类似，但如果键入^D 则打印一条消息
C,D,F,T	与 c，d，f，t 类似，但它从一个给定的目录中选择补全
(list)	从词列表中选择补全

范例 9-90

```
# p-type completions(positional completion)

1  > complete
   alias     'p/1/a/'
```

```
     cd        'p/1/d/'
     ftp       'p/1/( owl ftp.funet.fi prep.ai.mit.edu )'
     man       'p/*/c/'
2    > complete vi 'p/*/t/'
3    > complete vi
     vi 'p/*/t/'
4    > set autolist
5    > man fin[tab]     # Completes command names
     find    find2perl   findaffix  findsmb  finger
6    > vi b[tab]        # Completes only filenames, not directories
     bashtest binded bindings bindit
7    > vi na[tab]mes
8    > cd sh[tab]ellsolutions/
9    > set hosts = ( netcom.com 192.100.1.10 192.0.0.200 )
10   > complete telnet 'p/1/$hosts/'
11   > telnet net[tab]com.com
     telnet netcom.com
12   > alias m[tab]     # Completes alias names
     mc mroe mv
13   > ftp prep[tab]
```

说明

1. 不带参数的 complete 命令列出了所有的编程补全。后面的例子(第 2~11 行)将使用这些补全规则。

2. 这条规则规定，如果要在键入 vi 命令参数时使用 Tab 键补全，那么所有的参数必须是“t”类型(例如，纯文本文件)。

3. 以命令名作为参数的 complete 命令用于显示该命令的规则。这里显示了 vi 的补全规则。

4. 通过设置内置命令 autolist，所有可能的 Tab 键补全将自动被打印(均需按下 Ctrl+D)。

5. man 命令有一个编程补全：complete man 'p/1/c/'。这条规则规定，man 命令的第一个参数必须是一个命令。因为 c 定义其类型为命令补全。本例中，试着补全的是 man 的参数 fin(命令名)，因此，所有以 fin 开头的命令均被列出。

6. 因为将 vi 编辑器的补全编程针对的是文本文件，所以不对目录补全，只有文件名才会被补全。

7. 根据 vi 补全规则，无论传多少参数，仅有文本文件名才会被补全。

8. 当对内置命令 cd 的第一个参数执行文件名补全时，补全规则中规定，被补全的词必须为目录名。这个例子中的参数将扩展为目录名 shellsolutions。

9. 变量 hosts 被设置为 IP 地址或主机名的列表。

10. telnet 的补全规则规定：仅当位置 1 包含 hosts 变量设置的一个主机名时才执行补全。本例的类型为单词补全。

11. 执行 telnet 命令，以 net 开头的词在按下 Tab 键后被补全为前面设置的 hosts 变量中的一个主机名 netcom.com。

12. 如果用户键入词 alias，后跟一个单词，被扩展后的所有别名都将包含这个词，则将执行别名补全。此例类型为别名补全。



一种称为 c 类型的补全用于补全当前词中的模式。当前词指的是括在正斜杠中的模式。它规定，如果模式能够匹配，则所执行的补全将完成模式。

范例 9-91

```
# c-type completions
1  > complete
   stty    'c/-/(raw xcase noflsh)/'
   bash    'c/-no/(profile rc braceexpansion)/'
   find    'c/-/(user name type exec)/'
   man     'c/perl/(delta faq toc data modlib locale)/'
2  > stty -r[tab]aw
   stty -raw
3  > bash -nop[tab]rofile
   bash -noprofile
4  > find / -n[tab]ame .tcshrc -p[tab]rint
   find / -name .tcshrc -print
5  > man perlde[tab]lta
   man perldelta
6  > uncomplete stty
   > complete
   bash    'c/-no/(profile rc braceexpansion)/'
   find    'c/-/(user name type exec)/'
   man     'c/perl/(delta faq toc data modlib locale)/'
7  > uncomplete *
```

说明

1. 这些例子展示了 c 类型的补全。如果键入了第一对正斜杠中的模式，即键入了圆括号中的某个词的一个或多个字符并按下 Tab 键后，这个模式将被圆括号中列出的词补全。

2. 当键入了 stty 命令和一个长划号后，只要再键入一个 r 并按下 Tab 键，则这个词将被补全为-raw。圆括号括着的(raw xcase noflash)规则列表中的一个词可以用来完成补全。

3. 当键入了 bash 命令和-no 模式，如果再键入一个 p，并按下 Tab 键，则这个模式将被补全为-noprofile。执行的补全来自规则列表(profile rc braceexpansion)中的一个词。在这个例子中，结果输出为-noprofile。

4. 如果 find 命令的参数是一个长划号后面跟着 find 规则列表(user name type exec)中某个词的前一个或多个字符，则这个参数将被补全。

5. 键入 man 命令后，模式 perl 被补全为 perldelta，因为这个模式后跟的是列表(delta faq doc data modlib locale)中的一个词。

6. 内置命令 uncomplete 用于删除 stty 的补全规则。其他的补全规则被保留。

7. 以*为参数的内置命令 uncomplete 删除了所有的补全规则。

而 n 类型的补全则将对第一个词进行匹配并补全第 2 个词。

范例 9-92

```
# n-type completions(next word completion)
1  > complete
   rm   'n/-r/d/'
```

```
   find 'n/-exec/c/'
2  > ls -ld testing
   drwxr-sr-x  2  ellie  root   1024 Aug 29 11:02 testing
3  > rm -r te[tab]sting
```

说明

1. 本例展示的是 n 类型补全。如果键入了(当前词)第一对正斜杠中的词并匹配成功，则根据词类型将自动补全下一个词(在第二对正斜杠中)。complete 命令列出了两个 n 类型补全。一个是 rm 命令，一个是 find 命令。如果打开-r 开关执行 rm 命令，那么-r 后的词类型必须为目录才能进行补全。find 命令的规则是：如果给定了-exec 选项，要执行补全，则它后面的词必须为命令。

2. ls 命令的输出显示 testing 是一个目录。

3. rm 命令成功地执行了文件名补全，这是因为它尝试的补全对象是一个 testing 目录。如果 testing 是一个无格式文件，则补全将不会执行。

9.16 TC shell 拼写校正

TC shell 新增加了一种称为拼写校正的新特性，可以校正文件名、命令和变量中的拼写错误。如果使用 emacs 内置编辑器，可以使用拼写校正键 Meta-s 或 Meta-S(如果没有 Meta 键，可以使用 Alt 或 Esc 键)来校正拼写错误，或者使用 Meta-$来校正一整行。prompt 的值设为 prompt3 将显示拼写校正提示[9]。

如果使用内置编辑器 vi，设置内置变量 correct 后，shell 将提示您修改拼写。

表 9-18　correct 变量参数

参　数	作　用
all	对整个命令行进行拼写校正
cmd	对命令进行拼写校正
complete	补全命令

范例 9-93

```
1  > fimger[Alt-s]    # Replaces fimger with finger
2  > set correct=all
3  > dite
   CORRECT>date (y|n|e|a)? yes
   Wed Aug 8 19:26:27 PDT 2004
4  > dite
   CORRECT>date (y|n|e|a)? no
```

[9] 摘自 tcsh 帮助页："注意，拼写校正并不能保证像您所希望的那样运行，它目前还处于试验阶段。因此，非常期望得到您的建议或反馈意见。"



```
    dite: Command not found.
    >
5   > dite
    CORRECT>date (y|n|e|a)? edit
    > dite      # waits for user to edit and then executes command
6   > dite
    CORRECT>date (y|n|e|a)? abort
    >
```

说明

1. 通过按下 Alt 键(或 Esc 键)和 s，命令、文件名或变量的拼写可以被校正。如果使用内置编辑器 vi 则这样做无效。

2. 通过将 correct 设置为 all，tcsh 将试着校正命令行中所有的拼写错误。这个特性对 emacs 和 vi 按键绑定均有效。

3. 因为命令拼写错误，第 3 种提示符 prompt3 将在屏幕上显示"CORRECT>date (y|n|e|a)?"。如果用户希望进行拼写校正则键入字母 y，不希望则键入字母 n，如果希望编辑命令行则键入 e，或者键入 a 以中止整个操作。

4. 如果用户希望命令保持不变，则键入 n 即可。

5. 如果用户希望编辑校正，则键入 e，接着将提示用户修改或增强命令。

6. 如果校正不正确或是不希望的，则用户键入一个 a，拼写校正就会被中止。

9.17 TC shell 别名

别名是 TC shell 中用户定义的命令缩写。当命令有很多选项和参数或者命令的语法非常难记时，别名是很有用的。在命令行设置的别名不会被子 shell 继承。别名通常在.tcshrc 文件中设置。当启动新的 shell 时会执行.tcshrc 文件，因此，这个文件中设置的别名将被重置到新 shell 中。别名也可以传给 shell 脚本，但这样可能会导致潜在的移植性问题，除非直接在脚本中设置它们。

TC shell 有一些预置的别名，在您定义之前它们一直处于未定义状态。它们是：beepcmd，cwdcmd，periodic 和 precomd。这些别名将在表 9-19 tcsh 别名中列出并定义。

9.17.1 列出别名

内置命令 alias 列出设置的所有别名。首先显示的是别名，然后显示的是实际命令或它所代表的命令。

范例 9-94

```
> alias
apache   $HOME/apache/httpd -f $HOME/apache/conf/httpd.conf
co       compress
cp       cp -i
ls1      enscript -B -r -Porange -f Courier8 !* &
mailq    /usr/lib/sendmail -bp
```

```
mc      setenv MC '/usr/bin/mc -P !*'; cd $MC; unsetenv MC
mroe    more
mv      mv -i
uc      uncompress
uu      uudecode
vg      vgrind -t -s11 !:1 | lpr -t
weekly   (cd /home/jody/ellie/activity; ./weekly_report; echo
Done)
```

说明

alias 命令在第一列中列出了命令的别名(简称)，第二列是别名所代表的实际命令。

9.17.2 创建别名

alias 命令用于创建别名。第一个参数是别名的名称，即命令的缩写。该行的其他部分则是执行别名时实际要执行的命令。多个命令之间用分号间隔，命令中的空格和元字符则使用单引号引用。

格式

```
alias
alias aliasname command
alias aliasname 'command command(s)'
unalias aliasname
```

范例 9-95

```
1  > alias m more
2  > alias mroe more
3  > alias lf ls-F
4  > alias cd 'cd \!*; set prompt = "%/ > "'
5  > cd ..
6  /home/jody >  cd /          # new prompt displayed
   / >
7  > set tperiod = 60
   > alias periodic 'echo You have worked an hour, nonstop'
8  > alias Usage 'echo "Error: \!* " ; exit 1'
```

说明

1. 命令 more 的别名被设置为 m。
2. 命令 more 的别名被设置为 mroe，这主要是方便可不会拼写的情况。
3. 别名 lf 是 tcsh 内置命令 ls-F 的缩写。它与 ls –F 一样用于列出文件，但是速度更快。
4. 执行 cd 命令时，cd 的别名将导致 cd 进入作为参数的目录，然后将提示符重置为当前工作目录(%/)后跟字符串 “%/ >”。别名使用!*的方式与历史机制使用它们的方式相同。反斜杠用于在别名使用!*之前阻止历史机制先对其求值。\!*代表历史列表中最近使用命令的参数。因为有空白符，所以用引号将别名定义括起来。
5. 在 cd 命令切换到父目录后，提示符被扩展为当前工作目录(%/)和一个>符号[10]。

[10] 如果使用的 shell 是/bin/csh，当设置提示符时需使用$cwd 而不是%/。



6. 从提示符可以看出，新的目录是/home/jody，切换目录至根目录(/)后，新提示符再次出现。

7. 变量 tperiod 被设置为 60 分钟。别名 periodic 是一个预置别名。设置后，每 60 分钟，echo 语句就会被显示一次。

8. 在脚本中生成诊断消息并退出脚本时，别名是很有用的。范例 10-20 将展示别名的这种用法。

9.17.3 删除别名

unalias 命令用于删除别名。在别名前加一个反斜杠可以临时地关闭一个别名。

范例 9-96

```
1   > unalias mroe
2   > \cd ..
```

说明

1. unalias 命令从定义的别名列表中删除别名 mroe。
2. 仅在执行这个命令时，别名 cd 临时地被关闭。

9.17.4 别名循环

当一个别名定义引用了另外一个别名，而被引用的别名反过来又引用了原来的别名时，就出现了别名循环。

范例 9-97

```
1   > alias m more
2   > alias mroe m
3   > alias m mroe        # Causes a loop
4   > m datafile
    Alias loop.
```

说明

1. 别名为 m，别名定义为 more。每次使用 m 将执行 more 命令。
2. 别名为 mroe，别名定义为 m。如果键入 mroe，将调用别名 m 并执行 more 命令。
3. 这里是导致循环的地方。如果使用别名 m，它将调用别名 mroe，由于别名 mroe 再次引用 m，这就导致了一个别名循环。但不会有什么大问题，仅仅是得到一个出错的消息。
4. 使用了别名 m，它其实是一个循环。于是，m 调用 mroe，mroe 调用 m，接着 m 又调用 mroe，以此类推。但 TC shell 会及时发现问题并显示出错信息，这样循环就不会继续下去。

9.17.5 特殊的 tcsh 别名

如果已设置，则每个 TC shell 的别名(见表 9-19)都将在指定时间自动执行。且它们初始化时都未被定义。

表 9-19　tcsh 别名

别　名	作　用
beepcmd	当 shell 终端响铃时运行
cwdcmd	每次改变工作目录时运行
periodic	每隔 tperiod 中指定的分钟数运行一次 例如： >set tperiod=30 >alias periodic date
precmd	每次打印提示符之前运行，例如：alias precmd date

9.18　TC shell 作业控制

作业控制是 TC shell 中一个很强大的特性，它允许在前台或后台运行被称为作业的程序。通常情况下，在命令行键入的命令将在前台运行直到结束。如果有窗口程序，则作业控制就是不必要的，因为可以简单地打开另外一个窗口以启动一个新任务。但是，如果只有一个终端，那么作业控制将是一个非常有用的特性。表 9-20 是作业命令列表。

表 9-20　作业控制命令

命　令	含　义
jobs	列出当前运行的所有作业
^Z(Ctrl+Z 组合键)	暂停(挂起)作业，屏幕上出现提示符
bg	开始在前台运行挂起的作业
fg	将一个后台作业提到前台运行
kill	向某个作业发送 kill 信号
jobs 命令参数	
%n	作业号 n
%string	以字符串开头的作业名
%?string	包含字符串的作业名
%%	当前作业
%+	当前作业
%-	当前作业的前一个作业

9.18.1　jobs 命令与 listjobs 变量

tcsh 内置命令 jobs 将打印当前处于活动状态的程序，包括在后台运行和被挂起的作业。运行指的是作业正在后台执行，挂起则指的是作业被阻塞，即不处于执行状态。这两种情况下，终端都可以接收其他命令。如果在进程被阻塞时尝试退出 shell，屏幕上将会显示警



告“There are suspended jobs”。当立即再次执行退出时，shell 将直接终止挂起的作业并退出。如果希望在挂起作业时自动显示一条消息，可以设置 tcsh 内置变量 listjobs。

范例 9-98
(命令行)

```
1   > jobs
2   [1]  +          Suspended      vi filex
    [2]  -          Running        sleep 25
3   > jobs -l
    [1]  +   355    Suspended      vi filex
    [2]  -   356    Running        sleep 25
4   [2] Done                       sleep 25
5   > set listjobs = long
    > sleep 1000
    Press Ctrl+Z to suspend job
    [1]  + 3337    Suspended      sleep 1000
    >
6   > set notify
```

说明

1. jobs 命令列出当前活动的作业。

2. 标记[1]是第一个作业的编号，加号表明该作业不是最近被放到后台的作业。长划线表明该作业是最近放入后台的作业。suspended 意思是该作业用^Z 阻塞了，当前处于非活动状态。

3. -l 选项(long listing)显示作业编号和 PID。标记[2]是第二个作业的编号，在这个例子中，它是最新放入后台的作业。长划线表明这是最近的作业，即 sleep 命令位于后台运行。

4. sleep 运行 25 秒后，作业完成，屏幕上显示一条消息表明作业已经完成。

5. 如果将 tcsh 的变量 listjobs 设置为 long，则进程挂起时将打印作业的编号和进程的 id 号(9.20.1 节中的表 9-26 是 tcsh 内置变量列表)。

6. 如果有作业被阻塞，shell 通常在打印提示符之前给出通知。但是如果设置了 shell 变量 notify，shell 将在后台作业的状态发生任何变化后立即通知您。例如，如果您正在使用 vi 编辑器，此时一个后台作业结束，vi 窗口将立即显示下面一条消息：

```
[1]     Terminated                  sleep 20
```

9.18.2 前台与后台命令

fg 命令将一个后台作业提到前台运行，bg 命令在后台开始运行一个被阻塞的作业。如果希望选择一个特定的作业进行作业控制，使用一个百分号加作业编号作为 fg 和 bg 的参数。

范例 9-99

```
1   > jobs
2   [1] + Suspended                vi filex
    [2] - Running                  cc prog.c -o prog
3   > fg %1
    vi filex
```

```
    (vi session starts)
4   > kill %2
    [2] Terminated              c prog.c -o prog
5   > sleep 15
    (Press ^z)
     Suspended
6   > bg
    [1] sleep 15 &
    [1] Done   sleep 15
```

说明

1. jobs 命令列出当前运行的进程，即作业。

2. 第一个被阻塞的作业是 vi 会话，第二个作业是 cc 命令。

3. 编号为[1]的作业被提到前台，作业编号之前是一个百分号。

4. kill 是内置命令。默认情况下它向一个进程发送 TERM(终止)信号。参数是进程编号或 PID。

5. 按下^Z 以阻塞 sleep 命令。sleep 命令将不再占用 CPU，它在后台被挂起。

6. bg 命令将使后台最近的一个作业在后台恢复执行。sleep 程序开始为进程恢复执行进行倒计时。①

9.18.3 作业调度

内置命令 sched 允许您创建一个作业列表，它们可以在指定的时间被调度执行。不带参数的 sched 命令，将显示所有调度事件的编号列表。它将时间设置为 hh:mm(小时:分钟)的格式，其中小时可以是军用格式或 12 小时制的上午/下午格式。也可以使用一个+号从而将时间指定为基于当前时间的相对时间。如果使用-号，则这个事件将从列表中被删除⑫。

格式

```
sched
sched [+]hh:mm command
sched -n
```

范例 9-100

```
1   > sched 14:30 echo '^G Time to start your lecture!'
2   > sched 5PM echo Time to go home.
3   > sched +1:30 /home/ellie/scripts/logfile.sc
4   > sched
     1    17:47 /home/scripts/logfile.sc
     2    5PM echo Time to go home.
     3    14:30  echo '^G Time to start your lecture!'
5   > sched -2
    > sched
```

---

① 诸如 grep、sed 和 awk 之类的程序都有一个元字符集合用于模式匹配，称为正规表达式通配符。不要将它们与 shell 元字符混淆。

⑫ 摘自 tcsh 帮助手册“调度事件列表中的命令在命令指定的调度时刻到达之后，于第一次提示打印之前被执行。命令有可能会错过命令应执行的正确时间，但一个过期的命令将在下一次提示时执行”。




```
   1    17:47 /home/scripts/logfile.sc
   2    14:30  echo '^G Time to start your lecture!'
```

说明

1. 调度命令 sched 在 14:30 执行 echo 命令。界时将会发出一声蜂鸣(相当于按下 Ctrl+G 组合键)[13]并打印一条消息。

2. sched 命令将在下午 5 时调度执行 echo 命令。

3. 脚本 logfile.sc 将在从现在开始计时，于 1 小时 30 分钟后被调度执行。

4. sched 命令以编号顺序显示调度事件，最近调度的事件将第一个显示。

5. 以一个编号为参数，sched 将删除调度列表中此编号对应的作业。sched 的输出显示了原编号为 2 的作业已被删除。

9.19 在 TC shell 中显示变量的值

9.19.1 echo 命令

内置命令 echo 向标准输出打印它的参数。echo 命令允许使用众多的转义序列，它们可以被解释和显示为 tab、换行符、换页符等。表 9-21 列出了 echo 的选项和转义序列。

TC shell 支持 BSD 与 SVR4 两种风格的 echo 命令，允许使用内置变量 echo_style 修改 echo 命令的行为。例如，set echo_style=bsd。参见表 9-22 和 echo 命令帮助页。

表 9-21　echo 选项和转义序列

选　项	含　义
-n	取消输出行尾的换行符
转义序列	
\a	报警(响铃)
\b	退格符
\c	不带换行符打印行
\f	换页符
\n	换行符
\r	回车符
\t	制表符
\v	垂直制表符
\\	反斜杠
\nnn	ASCII 码为 nnn(八进制)的字符

[13] 要想输入 echo 语句中的^G 命令，也可以先按下 Ctrl+M 组合键，接着按下 Ctrl+V 组合键，最后再按 Ctrl+G 组合键。

表 9-22　echo_style 变量(SVR4 与 BSD)

系　统	作　用
bsd	如果第一个参数是-n，则换行符被取消
both	-n 与转义序列均有效(默认)
none	既不识别 sysv 也不识别 bsd
sysv	在 echo 字符串中扩展转义序列

范例 9-101

```
1   > echo The username is $LOGNAME.
    The username is ellie.
2   > echo "\t\tHello there\c"
              Hello there>
3   > echo -n "Hello there"
    Hello there$
4   > set echo_style=none
5   > echo  "\t\tHello there\c"
    -n \t\tHello there\c
```

说明

1. echo 命令在屏幕上显示它的参数。在 echo 命令执行之前，shell 首先进行变量替换。
2. 与 SVR4 版本的 echo 命令用法或 C 编程语言中支持转义序列相似，默认情况下，echo 命令也支持转义序列。>符号是 shell 的提示符。
3. 带-n 选项的 echo 命令所显示的字符串没有换行符。
4. echo_style 变量被赋值为 none。BSD －n 开关与 SVR4 转义序列均无效。
5. 字符串以新的 echo 风格显示。

9.19.2　printf 命令

GUN 版本的 printf 可用于格式化打印输出。它以与 C 语言 printf 函数相同的方式打印格式化字符串。格式化字符串中包含用以描述打印输出效果的指令。格式化指令由一个%加指示符(diouxXfeEgGcs)组成。其中%代表一个浮点数，%d 代表一个(十进制)数。

在命令行提示符处键入 printf --help 将会看到 printf 的指示符列表(参见表 9-23)以及它们的用法。键入 printf --version 可以知道当前使用的 printf 的版本。

表 9-23　printf 命令的格式化指示符

格式化指示符	含　义
\"	双引号
\0NNN	一个八进制字符，NNN 代表 0~3 之间的数字
\\	反斜杠
\a	报警或响铃
\b	退格符
\c	不再进一步输出



(续表)

格式化指示符	含　义
\f	换页符
\n	换行符
\r	回车符
\t	水平制表符
\v	垂直制表符
\xNNN	十六进制字符，NNN 代表 1~3 之间的数字
%%	单个百分号
%b	参数作为一个字符串，可以使用\转义字符

格式

```
printf 格式 [参数...]
```

范例 9-102

```
printf "%10.2f%5d\n" 10.5  25
```

范例 9-103

```
1   > printf --version
    printf (GNU sh-utils) 1.16
2   > printf "The number is %.2f\n" 100
    The number is 100.00
3   > printf "%-20s%-15s%10.2f\n" "Jody" "Savage" 28
    Jody                Savage              28.00
4   > printf "|%-20s|%-15s|%10.2f|\n" "Jody" "Savage" 28
    |Jody                |Savage         |     28.00|
```

说明

1. 打印了 printf 命令的 GNU 版本。它位于/usr/bin 目录下。

2. 将参数 100 打印为保留两位小数浮点数形式，这是由格式化字符串中的%.2f 设定的。与 C 语言不同，这里没有用于分隔参数的逗号。

3. 这个格式化字符串指定了 3 处需要进行转换的地方：第 1 处是%-20s(一个左对齐的包含 20 个字符的字符串)，第 2 处是%-15s(一个左对齐的包含 15 个字符的字符串)，最后一处是%10.2f(一个右对齐的包含 10 个字符的浮点数，其中有一个字符是小数点，最后两个字符是小数点后的两位数字)。每个参数应依次按照相应的%进行格式化。因此字符串“Jody”对应第一个%，字符串“Savage”对应第二个%，数字 28 对应最后一个%。竖线用于指定各个域的宽度。

9.19.3 花括号与变量

花括号将变量与后面的字符隔开。它们可用于在变量后连接一个字符串。

范例 9-104

```
1   > set var = net
```

```
    > echo ${var}
    net
2   > echo $varwork
    varwork: Undefined variable.
3   > echo ${var}work
    network
```

说明

1. 花括号包围着变量名，从而将变量与后面的字符隔开。
2. 变量 varwork 尚未定义。shell 显示一条出错消息。
3. 因花括号包围变量而被隔开的字符串将追加在变量值的后面。$var 被扩展，字符串 work 追加在其后。

9.19.4 大小写转换

一个特殊的历史修饰符可用于改变变量中字母的大小写。

表 9-24 TC shell 大小写修饰符

修 饰 符	作 用
:a	对单个词尽可能多地应用修饰符
:g	对每个词只应用一次修饰符
:l	将词中的第一个大写字母改为小写
:u	将词中的第一个小写字母改为大写

范例 9-105

```
1   > set name = nicky
    > echo $name:u
    Nicky
2   > set name = ( nicky jake )
    > echo $name:gu
    Nicky Jake
3   > echo $name:agu
    NICKY JAKE
4   > set name = ( TOMMY DANNY )
    > echo $name:agl
    tommy danny
5   > set name = "$name:agu"
    > echo $name
    TOMMY DANNY
```

说明

1. 当将:u 追加到这个变量，它的第一个字母变为大写。
2. 当将:gu 追加到这个变量，每个词的第一个字母均变为大写。
3. 当将:agu 追加到这个变量，所有的字母均变为大写。
4. 当将:agl 追加到这个变量，所有的字母均变为小写。
5. 变量被重置，列表中所有字母大写。



9.20 TC shell 内置命令

不同于存储在物理磁盘上的 UNIX/Linux 可执行命令，内置命令是 C/TC shell 内部代码的一部分，它们从 shell 内部执行。如果内置命令作为管道除最后一部分以外的一部分，它将在子 shell 中执行。tcsh 有一条内置命令 builtins，可用来列出所有的内置命令。

范例 9-106

```
1  > builtins
:          @           alias      alloc      bg         bindkey    break
breaksw    builtins    case       cd         chdir      complete   continue
default    dirs        echo       echotc     else       end
endsw      eval        exec       exit       fg         filetest   foreach
glob       goto        hashstat   history    hup        if         jobs
kill       limit       log        login      logout     ls-F       nice
nohup      notify      onintr     popd       printenv   pushd      rehash
repeat     sched       set        setenv     settc      setty      shift
source     stop        suspend    switch     telltc     time       umask
unalias    uncomplete  unhash     unlimit    unset      unsetenv   wait
where      which       while
```

表 9-25 tcsh 内置命令与它们的含义

命 令	含 义
:	解释空命令，但不执行任何动作
alias [name [wordlist]]	命令的别名。不带参数将打印所有的别名；以一个名字为参数则打印该名字的别名；以一个名字和词列表为参数则将设置别名
alloc	显示所获取的动态内存的大小，分为已使用的内存和空闲内存。因系统而异
bg [%job] %job &	在后台运行当前的或指定的作业 内置命令 bg 的同义字
bindkey [-l|-d|-e|-v|-u] (+) bindkey [-a] [-b] [-k] [-r] [--] key bindkey [-a] [-b] [-k] [-c|-s][--] key command	不带选项时，第一种形式列出所有的快捷键及其对应的编辑器命令；第二种形式列出快捷键所对应的编辑器命令；第三种形式将编辑器命令绑定到按键。选项包括： -a 列出或更改二选一键映射表中的快捷键。这种键映射表用于 vi 命令模式 -b 这个键可以解释为一个控制键，写作^字符(例如，^A)或 C-字符(例如，C-A)；解释为一个元字符，写作 M-字符(例如，M-A)；解释为一个功能键，写作 F-字符串(例如，F-字符串)；或者解释为一个扩展前缀键，写作 X-字符(例如，X-A) -c 这个命令被解释为内置命令或外部命令，而不是编辑器命令 -d 对默认编辑器，将所有的按键绑定为标准绑定

(续表)

命令	含义
	-e　将所有的按键绑定到标准的 GNU 类 emacs 绑定 -k　将该键解释为一个符号方向键名字，可以是“向下”、“向上”、“向左”或“向右”中的一个 -l　列出所有的编辑器命令和对它们的简短描述 -r　删除按键绑定。请注意：bindkey -r 并不将按键绑定到 self-insert 命令(参照该项)，它将彻底取消按键绑定 -s　这个命令等同于字面字符串，当按键被键入时，它被当作终端输入。命令中的快捷键由它们自己重新解释，这种解释可以持续到 10 级 -u　(或者任何无效选项)将打印一条使用消息。这个键可能是一个单字符或者一个字符串。如果某个命令绑定到一个字符串，那么该字符串的第一个字符绑定到 sequence-lead-in，整个字符串绑定到命令 -v　将所有的按键绑定到标准的类 vi(1)绑定 --　强制跳出选项处理，因此即使下一个词以连字号开头，它也将被当作一个按键 按键上的控制字符可以作为文字(在它们前面加上编辑器命令 quoted-insert(通常绑定到^V)就可以将它们键入)或者写成^符号风格(例如，^A)。Delete 可以写作^?(脱字符加上问号)。键和命令可以包含下述反斜杠转义序列(使用 System V echo(1)的风格)： \a　响铃 \b　退格符 \e　ESC 键 \f　换页符 \n　换行符 \r　回车符 \t　水平制表符 \v　垂直制表符 \nnn　对应八进制数字 nnn 的 ASCII 字符 \　如果它有反斜杠(\)或脱字符(^)，则使它们的特殊意义失效
break	跳出最内层的 foreach 循环或 while 循环
breaksw	从 switch 中跳出，在 endsw 之后恢复执行
builtins	打印所有的内置命令名
bye	内置命令 logout 的同义词。只有这样编译它才可用，参见 shell 变量 version
case label:	switch 语句的标号



(续表)

命　令	含　义
cd [dir]	将 shell 的工作目录切换到 dir。如果没有带参数，则切换到用户主目录
cd [-p] [-l] [-n|-v] [name]	如果给出了目录名，则将 shell 工作目录切换到该目录下。如果没有，则切换到主目录。如果 name 参数是"-"，则将其解释为前一个工作目录。-p 与 dirs 一样，用来打印最后的目录栈。-l、-n 和-v 标志的用法在 cd 上和在 dirs 上是一样的，它们都包含了-p
chdir	内置命令 cd 的同义词
complete [word/pattern/list[:select]/ [[suffix]/]...]]	不加参数时，列出所有的补全。以命令为参数，列出该命令的所有补全。以命令和词为参数，则是定义补全(参见 9.15.4 节"编程补全")
continue	从包围该语句的最近的 while 循环或 foreach 循环处继续执行
default:	标示着 switch 语句的 default 状态。default 应该在所有其他状态标记的后面
dirs [-l] [-n|-v] dirs –S|-L [filename] dirs -c	第一种形式打印目录栈。栈顶元素在左，第一个目录是当前目录。输出中的-l、~或~name 被显式地扩展为主目录或用户主目录的路径名 (+) –n　显示的项在到达屏幕边界处时自动换行 (+) –v　每个目录项占一行进行打印，前面是目录在栈中的位置 -s　第二种形式将目录栈以一系列的 cd 命令和 pushd 命令的形式保存到文件 filename 中 -L　shell 源文件名，可能是保存的目录栈文件 无论是哪种形式，如果没有给出文件名则使用 dirsfile，如果没有设置 dirsfile 则使用~/.cshdirs 文件 使用-c 的第三种形式清空目录栈
echo [-n] list	将 list 中的词写入 shell 的标准输出，各词之间以空格分隔。除非使用-n 选项，否则输出将以一个换行符结束
echo [-n] word ...	将每个词写入到 shell 的标准输出，词之间以空格分隔，并以换行符结束。可以通过设置 shell 变量 echo_style 以模仿 BSD 或 System V 版本的 echo 中的标志或转义序列
echotc [-sv] arg ...	以 arg 检测终端的功能(参见 term-cap(5))。例如，echotc home 将光标移动到行的 home 位置。如果 arg 是 baud、cols、lines、meta 或 tabs，则打印该功能的值。使用-s，则不具备的功能返回空字符串，从而不导致出错。使用-v，则打印更为详细的信息
else if (expr2) then	参见 10.6 节"条件结构与流控制"
else end endif endsw	参见下面关于 foreach、if、switch 和 while 语句的描述

(续表)

命　令	含　义
end	当 expr 的值非 0 时执行 while 关键字与所对应的 end 关键字之间的命令。while 和 end 必须单独为一行。break 与 continue 可以用来提前终止或继续执行循环。如果输入的是终端，则用户将在第一次执行循环(如 foreach)时得到提示
eval arg ...	将参数作为 shell 的输入，在当前 shell 环境中执行所产生的命令。这通常用于执行由命令或变量替换(替换前已进行了解析)产生的结果命令
eval command	运行 command，以其结果作为 shell 的标准输入，执行 shell 产生的命令。这通常用于执行由命令或变量替换(替换前已进行了解析，例如 eval 'tset –s options')所产生的结果命令
exec command	在当前 shell 中执行 command 直到结束
exit [(expr)]	以状态变量值或 expr 指定的值退出 shell
fg [%job] %job	将当前或指定作业 job 移至前台执行 内置命令 fg 的同义词
filetest –op file ...	对每个文件应用 op(文件查询操作)并返回一个由空格隔开的列表
foreach name (wordlist) ... end	参见 10.7.4 节“循环控制命令”
foreach var (wordlist)	参见 10.7.1 节“foreach 循环”
getspath	打印系统执行路径
getxvers	打印实验版本的前缀
glob wordlist	对 wordlist 执行文件名扩展。与 echo 类似，但不识别转义符(\)。输出时各词之间以空字符分隔
goto label	参见 10.6.14 节“goto 命令”
goto word	word 是一个文件名，命令替换后成为一个字符串，从而变成“label.”的格式。shell 尽可能地对其输入进行上溯，搜索含有“label.”格式的行(可能前面还有些空白或 tab 键)，然后从该行开始继续执行
hashstat	打印一行统计信息，指示内部 hash 表在定位命令(以及避免 exec)的命中率。exec 是指对 hash 函数可能命中的路径以及路径中的不以反斜杠开头的每一部分都进行尝试
history [-hTr] [n] history –S|-L|-M [filename] history –c	第一种格式打印历史事件列表。如果给定 n，则只有最近的 n 个事件被打印或保存。使用-h，则打印不带编号的历史列表。如果指定了-T，则时间戳也以说明的格式打印出来(这可以用来产生适合于 history –L 或 source –h 加载的文件)。使用-r，则打印时以最近事件优先的原则进行排序(参见 9.14.2 节“TC Shell 命令行历史”)使用-c，则清空历史列表



(续表)

命　　令	含　　义
hup [command]	以命令名为参数，则执行这个命令的过程中，一旦收到挂起信号，它将退出。并且当 shell 退出时会给它发送一个挂起信号。注意命令可能会以自己的挂起信号处理机制覆盖 dup。如果不带参数(仅在脚本中允许这种情况)，则如果脚本的其余部分如果收到挂起信号 shell 将退出
if (expr) command	如果 expr 为真，则命令将会被执行。命令上的变量替换与 if 命令其他部分一样，在此之前执行。命令必须是一个简单命令，不能是别名、管道、命令列表或者由圆括号括起来的命令列表，但是可以有参数。即使 expr 为假，命令不被执行，输入/输出重定向也会发生。这算是一个 bug
if (expr) then ... else if (expr2) then ... else ... endif	如果指定的 expr 为真，则第一个 else 之前的命令将被执行。否则如果 expr2 为真，第一个 else 和第二个 else 之间的命令将被执行，以此类推。else if 的数量不受限制，但只需要一个 endif，else 部分同样是可选的(else 与 endif 必须出现在行的开始处，if 必须单独成行或位于 else 的后面)
inlib shared-library ...	将每个共享库加入当前环境，从而无法删除共享库(仅限 Domain/OS 系统)
jobs [-l]	列出作业控制下的所有活动的作业。使用-l 参数，除常规信息外还列出 ID
kill [-sig] [pid] [%job] ... kill –l	向指定 ID 的作业或当前作业发送默认的 TERM(终止)信号或指定的信号。信号由数字或名字指定，没有默认的方式。键入 kill 将不会向当前作业发送任何信号，如果发送的是 TERM(终止)信号或 HUP(挂起信号)，则同时也会向作业或进程发送一个 CONT(继续)信号。使用-l 参数将列出所有可以被发送的信号
limit [-h] [resource [max-use]]	对当前进程及其派生进程使用资源的数量进行限制。每个进程使用指定 resource 的数量不超过 max-use 的值。如果省略了 max-use 则打印出当前的 limit，如果省略了 resource 则显示所有的 limit。使用-h 选项，可以用硬限制取代当前的限制。硬限制是在当前限制上强加上一个最高限度。只有超级用户才有权增加硬限制的值。resource 可以是下面的一种：cputime，每个进程的最大 CPU 时间片；filesize，单个文件允许的最大值；datasize，每个进程最大的数据大小(包括栈)；stacksize，每个进程栈的最大值；coredump，信息转储的最大值；descriptors，文件描述符的最大值
log	打印 shell 变量 watch，报告 watch 中指定的用户有哪些已经登录，不考虑他们的登录时间。请参见 watchlog
login	终止登录 shell，用/bin/login 的实例来取代它

(续表)

<table>
<tr><th>命　令</th><th colspan="2">含　义</th></tr>
<tr><td>login [username|-p]</td><td colspan="2">不处理.logout 文件，终止一个登录 shell 并调用 login(1)。如果省略了 username，则 login 提示输入用户名。使用-p 参数可以保存当前环境(变量)</td></tr>
<tr><td>logout</td><td colspan="2">终止登录 shell</td></tr>
<tr><td rowspan="15">ls–F [-switch ...] [file ...]</td><td colspan="2">以类似 ls –F 的方式列出文件，相比之下速度更快。它识别出特定文件的类型并用一个特别的字符列出。</td></tr>
<tr><td>/</td><td>目录</td></tr>
<tr><td>*</td><td>可执行</td></tr>
<tr><td>#</td><td>块设备</td></tr>
<tr><td>%</td><td>字符设备</td></tr>
<tr><td>|</td><td>命名管道(仅支持命名管道的系统)</td></tr>
<tr><td>=</td><td>套接字(仅支持套接字的系统)</td></tr>
<tr><td>@</td><td>符号链接(仅支持符号链接的系统)</td></tr>
<tr><td>+</td><td>隐藏目录(仅适用于 AIX 系统)或上下文相关(仅适用于 HP-UX 系统)</td></tr>
<tr><td>:</td><td>网络专用(仅适用于 HP-UX 系统)</td></tr>
<tr><td colspan="2">如果设置了 shell 变量 listlinks，则能识别出符号链接更详细的内容(当然，仅在支持它们的系统上)</td></tr>
<tr><td>@</td><td>非目录的符号链接</td></tr>
<tr><td>></td><td>目录的符号链接</td></tr>
<tr><td>&</td><td>无目的的符号链接</td></tr>
<tr><td colspan="2">内置命令 ls-F 可以根据文件类型和扩展名使用不同的颜色列出文件</td></tr>
<tr><td>migrate [-site] pid|%jobid ...
migrate –site</td><td colspan="2">第一种格式将进程或作业迁移到指定位置或由系统路径决定的默认位置。第二种格式等价于 migrate –site $$。它将当前进程迁移到指定位置。迁移 shell 本身会导致不可预料的行为，因为 shell 不愿失去 tty</td></tr>
<tr><td>@
@ name = expr
@ name[index] = expr
@name++|--
@name[index] ++|--</td><td colspan="2">第一种格式打印所有的 shell 变量。第二种格式将 expr 的值赋给 name。第三种格式将 expr 的值赋给 name 的第 index 个部分，name 和它的第 index 个部分都必须存在。第四种和第五种格式增加(++)或减少(--)名字或它的第 index 个部分</td></tr>
<tr><td>newgrp [-] group</td><td colspan="2">等价于 exec newgrp，参见 newgrp(1)。仅当 shell 这样编译后才可使用，参见 shell 变量 version</td></tr>
<tr><td>nice [+number] [command]</td><td colspan="2">将 shell 的调度优先级设置为 number，如果没有参数 number，则设为 4。如果有参数 command，则将以适当的优先级运行 command。number 数值越大，进程获得的 CPU 时间越少。超级用户可以用 nice –number 将优先级设为负值。command 总是在子 shell 中执行，当使用语句时自然地加上对命令所作的限制</td></tr>
</table>



(续表)

命　令	含　义
nohup [command]	运行 command 时，HUP(挂起)信号被忽略。如果不带参数，则脚本剩余所有部分都将忽略 HUP
notify [%job]	当前或指定作业的状态发生变化时异步地通知用户
onintr [- | label]	控制 shell 对中断的处理。如果不带参数，onintr 恢复 shell 对中断的默认处理(shell 终止脚本并返回到终端命令输入状态)。使用减号作为参数，shell 将忽略所有的中断。如果使用 label 参数，则在收到中断或子进程因为被中断而终止时，shell 执行 goto label
popd [+n]	弹出目录栈并进入到新的顶层目录。目录栈中的元素从 0 开始编号，顶层目录是第一个。使用+n 参数，将删除目录栈中第 n 个目录项
printenv [name]	打印所有环境变量的名称和值，或者使用 name 参数，打印该环境变量名的值
pushd [+n | dir]	将一个目录压入目录栈。如果不带参数，则交换最顶层的两个目录项。使用+n 参数，将第 n 个目录项移至栈顶并进入到该目录。使用 dir 参数，将当前工作目录压栈并进入到 dir 目录
rehash	为新增加的命令重新计算列于 path 变量上所有目录的内部哈希表
repeat count command	重复执行 count 次 command
rootnode //nodename	将 rootnode 改为//nodename，所以/将被解释为//nodename(仅用于 Domain/OS 系统)
sched sched [+]hh:mm command sched -n	第一种格式打印调度事件列表。可以设置 shell 变量 sched 以定义调度事件列表打印的格式。第二种格式向调度事件列表增加一条命令
set set name ... set name=word ... set [-r] [-f|-l] name=(wordlist) ... (+) set name[index]=word ... set –r set –r name set –r name=word ...	第 1 种格式的命令将打印所有 shell 变量的值。多于单个词的变量将以一个圆括号括着的词列表方式打印。第 2 种格式将 name 设置为空字符串。第 3 种格式将 name 设置为单个词。第 4 种格式将 name 设置为 wordlist 中的词表。在所有情况下其值为命令且文件名被扩展。如果指定了-r，则这个值被设为只读。如果指定了-f 或-l，则仅设置唯一的词并保持它们的顺序。-f 代指词的第一次出现，-l 代指词的最后一次出现。第 5 种格式将 name 的第 index 个部分设为 word，这个部分必须已经存在。第 6 种格式仅列出所有只读的 shell 的变量名。第 7 种格式将 name 设置为只读，无论它是否赋值。第 8 种格式与第 3 种格式相同，但同时将 name 设置为只读
set [var [= value]]	参见 9.10 节“变量”
setenv [VAR [word]]	参见 9.10 节“变量”。最常用的环境变量是 USER，TERM 和 PATH，自动地从 csh 变量 user，term 和 path 中导入并导出到这些变量中，对它们无需使用 setenv。另外，无论 csh 变量 cwd 何时发生变化，shell 都立即使用它来设置环境变量 PWD
setenv [name [value]]	不带参数时，打印所有环境变量的名字和值。给出 name 时，将环境变量 name 设置为 value 或设为空字符串

(续表)

命　令	含　义
setpath path	等价于 setpath(1)(仅用于 Mach)
setspath LOCAL|site|cpu ...	设置系统执行路径
settc cap value	通知 shell 使其相信终端能力 cap(在 termcap(5)中定义)的值已为 value，不用再做检测。终端用户需要使用 settc xn no 以使到最右列时能够正确地进行自动换行
setty [-d|-q|-x] [-a] [[+|-]mode]	控制 shell 不能改变的 tty 模式。-d、-q 或-x 分别通知 setty 作用在 edit,quote 或 execute 的 tty 模式集合上。没有-d、-q 或-x 参数时，使用 execute。没有其他参数时，setty 列出可选择集合中确定为开启(+mode)或关闭(-mode)的模式。可用的模式以及显示的内容因系统而异。使用-a 参数，将列出所选集合中所有的 tty 模式，无论它们是否是确定的。使用+mode,-mode 或 mode 分别将模式开启、关闭以及从所选集合中删除。例如，setty +echok echoe 将 echok 模式开启，当执行命令时允许命令打开或关闭 echoe 模式
setxvers [string]	将实验版本的前缀设置为 string 或删除(当 string 被省略时)
shift [variable]	将 argv 或 variable(如提供)的部分左移，丢弃第一部分。如果 variable 不是集合或者是空值，则产生错误
source [-h] name	从 name 读取命令。source 命令可以嵌套，但是如果嵌套太深，可能会致使 shell 描述符不够用。嵌套中任何一级的源文件出错将终止所有的 source 命令。一般通过重新执行.login 或.cshrc 文件以确保当前 shell 中的变量已经被设置。例如，shell 不能创建的子 shell(fork)。使用-h 参数，可以将 filename 中的命令加入到历史列表中，而不用执行它们
stop [%job] ...	阻塞当前或指定的后台作业
suspend	阻塞 shell 运行，就如同使用^Z 向 shell 发送阻塞信号一样。它常用于阻塞以 su 命令启动的 shell
switch (string)	参见 10.6.21 节“switch 命令”
telltc	列出所有终端能力的值(参见 termcap(5))
time [command]	不带参数，将打印出当前 shell 以及其派生 shell 的运行总时间，使用可选参数 command，则将执行 command 并打印出该命令的执行总时间
umask [value]	显示文件创建掩码。使用 value 参数，则将设置文件创建掩码。八进制的 value 与 666 进行异或将得到新创建文件的权限，与 777 进行异或将得到新创建目录的权限。不能直接通过 umask 设置 权限
unalias pattern	删除所有名字与 pattern 匹配的别名。unalias *将删除所有的别名。即使没有删除任何别名，也不算是一个错误
uncomplete pattern	删除所有名字与 pattern 匹配的补全。umcomplete *删除所有的补全
unhash	禁用内部 hash 表



(续表)

命　令	含　义
universe universe	将 universe 设置为 universe(仅用于 Masscomp/RTU)
unlimit [-h] [resource]	删除对 resource 资源的限制。如果没有指定任何 resource，则删除所有的资源限制。使用-h 参数，则删除相应的硬限制。只有超级用户才被允许这样做
unsetenv pattern	删除名字与 pattern 匹配的环境变量。unsetenv *删除所有的环境变量(最好不要这样做)。如果没有删除任何环境变量，也不会导致错误
unsetenv variable	删除环境变量 variable。与 unset 一样，这里并不执行模式匹配
@ [var=expr] @ [var[n] =expr]	不带参数时，显示所有 shell 变量的值。如果使用变量 var 或它的第 n 词为参数，则将其设置为 expr 的值
ver [systype [command]]	不带参数时，打印 SYSTYPE。以 systype 为参数，将 SYSTYPE 设置为 systype。以 systype 和 command 为参数，则将会以 systype 执行 command。systype 可以为 bsd4.3 或 sys5.3(仅用于 Domain/OS)
wait	在出现提示符前，等待后台作业结束(或等待中断)
warp universe	将 universe 设置为 universe(仅用于 Convex/OS)
watchlog	内置命令 log 的另一个名字。仅当 shell 以此方式编译时才可用。参见 shell 变量 version
where command	报告所有已知的 command 的实例。包括别名、内置命令和路径中的可执行文件
which command	显示在替换、路径搜索之后 shell 将执行的命令。这个内置命令同 whick(1)类似，但它能正确报告 tcsh 别名和内置命令，且速度快 10 到 100 倍
while (expr)	参见 10.7.4 节“循环控制命令”

9.20.1 特殊的内置 T/TC shell 变量

内置 shell 变量对 shell 有特殊的意义，它们用来改变和控制 shell 命令的行为。它们是局部变量，因此为了将其传递并影响子 shell，大部分 TC shell 内置变量在.tcshrc 文件中设置，C shell 内置变量在.cshrc 文件中设置。

当 shell 启动时，它自动设置下列变量：addsuffix，argv，autologout，command，echo_style，edit，gid，group，home，loginsh，oid，path，prompt，prompt2，prompt3，shell，shlvl，tcsh，term，tty，uid，user 和 version。除非用户决定改变这些变量，否则它们将一直固定不变。shell 还记录并改变需要定期更新的特殊变量，例如 cwd，dirstack，owd 和 status。用户退出时，shell 将设置 logout 变量。

有些局部变量对应一个同名的环境变量。如果用户环境中的变动影响了这两个变量中

的任何一个，shell 将同步更新另外一个局部变量或环境变量[14]，以使得这二者始终保持一致。这类交叉匹配变量的例子有 afuser，group，home，path，shlvl，term 和 user(尽管 cwd 和 PWD 有相同的含义，但它们不属于交叉匹配。虽然 path 和 PATH 变量的语法不同，但它们在一方发生变化时将自动进行交叉匹配)。

表 9-26　特殊的 C/TC shell 变量[15][16]

变　量	含　义
addsufix (+)	用于文件名补全。如果文件匹配成功，则通过在目录后加斜线以及在普通文件后加空格来进行补全。默认设置
afsuser (+)	如果设置，则 autologout 的自动锁特性将使用该值取代本地用户名进行 Kerberos 认证
ampm (+)	如果设置，则所有的时间使用 12 小时制，上午/下午的格式进行显示
argv	shell 命令行参数数组，也使用$1、$2 等形式表示
autocorrect (+)	在每次进行文件名、命令或变量补全之前首先调用拼写检查器
autoexpand (+)	如果设置，则每次进行补全之前自动调用历史扩展编辑器命令
autolist (+)	如果设置，则每次有多种补全方式时就列出所有的可能，如果设置为 ambiguous，则只有当无法再进行补全时才列出所有的可能
autologout (+)	它的参数是自动注销前系统处于不活动状态的时间(以分钟计)；第二个可选参数是自动锁屏前等待的时间(以分钟计)
backslash_quote (+)	如果设置，则反斜杠将总是引用自身、单引号和双引号
cdpath	一个目录列表。当子目录不在当前目录时，cd 命令在这个列表中搜索
color (+)	为内置命令 ls-F 打开色彩显示，向 ls 传递- -color=auto
complete (+)	如果设置为 enhance，则补全忽略大小写，将句点、连字符和下划线当作词分隔符，并认为连字符和下划线等价
correct (+)	如果设置为 cmd，则所有命令自动执行拼写更正；如果设置为 complete，则所有命令自动被补全；如果设置为 all，则整个命令行被更正
cwd	当前工作目录的路径名全称
dextract (+)	如果设置，则 pushd +n 从目录栈中取出第 n 个目录，而不是通过轮换方式将其移至栈顶
dirsfile (+)	命令 dirs –S 和 dirs –L 搜索历史文件的默认位置。如果被取消，则使用~/.cshdirs 文件
dirstack (+)	目录栈中所有目录的列表
dunique (+)	pushd 命令不允许目录栈中有相同的目录项
echo	如果设置，每个命令及其参数在执行前首先回显，该选项使用命令行选项-x 进行设置

⑭ 除非该变量是只读的，从而二者之间没有同步，否则一定会同步。
⑮ 带(+)的变量是 TC shell 特有的，其他的则内置于 TC shell 和 C shell 中。
⑯ 源自 tcsh 帮助手册中的描述。



(续表)

变　量	含　义
echo-style (+)	设置 echo 命令的风格。设置为 bsd 时，如果第一个参数为-n 则不回显换行符；如果设置为 sysv，则将识别出 echo 字符串中的反斜杠转义序列；如果设置为 both，则同时识别-n 标志和反斜杠转义序列；默认情况下，如果不设置，则都不识别
edit (+)	为交互式 shell 设置命令行编辑器，默认设置
ellipsis (+)	如果设置，则%c，%.和%C 提示符序列(参见 9.13.2 节"Shell 提示符")使用省略号而不是/<skipped>来代表被省略的目录
fignore	列出补全时忽略的文件名后缀
filec (+)	在 tcsh 中，这个变量被忽略，从而总是使用补全功能。如果在 csh 中设置了这个变量，则使用文件名补全
gid (+)	用户实际的组 ID 号
group (+)	用户组名称
hardpaths	如果设置，则目录栈中的路径名中不能含有符号链接
histchars	一个字符串值，它决定历史替换所使用的字符。它的第一个字符用作历史替换字符，以取代默认的字符'！'；它的第二个字符用于在快速替换中取代字符'^'
histdup (+)	控制对历史列表中同名项的处理方式。可设为 all(删除所有的同名项)、prev(如果当前命令与前面的命令同名，则删除当前命令)或 erase(如果当前事件与较早某个事件相同，则删除较早事件并将当前事件插入到历史列表中去)
histfile (+)	命令 history –S 和 history –L 搜索的是历史文件的默认位置。如果被取消，则使用~/.history 文件
histlit (+)	按字面意义向历史列表输入事件，也就是说，不使用历史替换进行扩展
history	第一个词指的是需要保存的历史事件的数目。可选的第二个词(+)指的是打印历史的格式
home	用户主目录，等价于~
ignoreeof	如果用户按下 Ctrl+D 组合键，则打印" Use "exit" to leave 退出 tcsh"。这样可以防止无意间注销
implicitcd (+)	如果设置，一旦键入某个目录名，则 shell 将其看作为进入该目录的命令，从而转换到该目录
inputmode (+)	如果设置为 insert 或 overwrite，则在每行的起始位置将编辑器置为输入模式
listflags (+)	如果设置为 x、a、A 或它们的任意组合(如 xA)，这些值将作为标志传给 ls –F，实际运行的是 ls –xF、ls –Fa、ls –FA 或标志的其他任意组合
listjobs (+)	如果设置，当某个作业挂起时，所有的作业都被列出。如果设置为"long"则列表的格式为长整型
listlinks (+)	如果设置，内置命令 ls –F 将显示每个符号链接所指向的文件类型
listmax (+)	在未明确指定的情况下，list-choices 编辑器命令将列出最大项数
listmaxrows (+)	在未明确指定的情况下，list-choices 编辑器将列出项的最大行数
loginsh (+)	登录 shell 时设置。在 shell 内部设置和取消它的两种操作都无效。参见 shlvl

(续表)

变　量	含　义
logout (+)	在正常注销前设置为 normal，自动注销前设置为 automatic，如果 shell 被中止信号挂起则设置为 hangup
mail	用于检查最近邮件的文件或目录的名字，如果有新邮件到达，则 10 分钟后将打印“You have new mail”
matchbeep (+)	如果设置为 never，则补全操作不再发出蜂鸣声；如果设置为 nomatch，则补全操作仅在当前无匹配时发出蜂鸣声；如果设置为 ambiguous，则当有多个匹配时发出蜂鸣声
nobeep	禁止发出蜂鸣声
noclobber	防止使用重定向(例如 ls > file)时意外删除文件的安全措施
noglob	如果设置，当使用通配符时禁止文件名替换和目录栈替换
nokanji (+)	如果设置则 shell 支持 Kanji(参见 shell 变量 version)，要使用元键则必须禁用它
nonomatch	如果设置，则当文件名替换或目录栈替换不能匹配现有的任何文件时，它们本身不做任何改变，而不是导致一个错误
nostat (+)	在补全操作时，一列目录(或匹配目录的 glob 模式)不应该被统计。这通常用于将需要大量时间进行统计的目录排除在外
notify	如果设置，则 shell 异步通知作业的完成，而无需再等到出现提示符
oid (+)	用户的实际的组 ID(仅用于 Domain/OS)
owd (+)	旧的或先前的工作目录
path	用于搜索可执行命令的一列目录。path 由 shell 在启动时从 PATH 环境变量来设置，如果 PATH 不存在，则其默认值与系统相关，一般是/usr/local/bin、/usr/bsd、/bin 和/usr/bin 之类的目录
printexitvalue (+)	如果设置，当交互程序以非 0 值退出时，shell 打印退出状态
prompt	在终端读取命令之前显示的字符串，可能包含一些特殊格式序列(参见 9.13.2 节“Shell 提示符”)
prompt2 (+)	在 while 循环和 loop 循环以及以‘\’结尾的行中使用的提示字符串。可以使用 prompt 中相同的格式序列。注意变量%R 的含义。在交互式 shell 中默认设置为%R?
prompt3 (+)	确认自动拼写更正时使用的字符串。可以使用 prompt 中相同的格式序列。注意变量%R 的含义。在交互式 shell 中默认设置为 CORRECT>%R(y|n|e|a)?
promptchars (+)	如果设置(一个由两个字符组成的字符串)，提示符 shell 变量中的%#格式序列将被替换，第一个字符用于普通用户，第二个用于超级用户
pushtohome (+)	如果设置，则不带参数的 pushd 执行 pushd ~命令，就如同 cd 命令一样
pushdsilent (+)	如果设置，pushd 和 popd 命令将不打印目录栈
recexact (+)	如果设置，命令补全即使存在更长的匹配也只进行精确匹配
recognize_only_executables (+)	如果设置，打印命令仅显示相应路径下的可执行文件
rmstar (+)	如果设置，则在执行 rm *命令之前用户将收到提示



(续表)

变　量	含　义
rprompt (+)	设置当提示符在屏幕左侧显示后，在屏幕右侧打印的字符串(命令输入后)。它识别与提示符相同格式的字符。它将在合适的时机显示或消失以确保命令输入不受影响。除非提示符、输入的命令与它可以在第一行组合，否则它将不会显示。如果没有设置 edit，则 rprompt 将在提示符之后，命令输入之前显示
savedirs (+)	如果设置，则 shell 退出前先执行 dirs –S
savehist	如果设置，则 shell 在退出前执行 history –S。如果第一个词为数字，则至多保存所设置的行数(这个数字必须小于或等于历史记录)。如果第二个词设为 merge，则将历史列表与当前历史文件合并，而不是替换(如果存在)它。然后使用 timestamp 排序从而就可以得到最近的事件
sched (+)	内置命令 sched 打印可调度事件的格式。如果没有显式地给出，则使用%h\t%T\t%R\n。序列格式与 prompt 显示的相同，注意%R 变量的含义
shell	shell 存在于这个文件。它用于派生 shell 来解释系统不能够执行的可执行文件(参见关于内置命令和非内置命令执行的介绍)。初始化为 shell 的主目录(与系统相关)
shlvl (+)	shell 嵌套数。登录后重置为 1。参见 loginsh
status	最后执行的命令返回的状态。如果是非正常结束，那么返回状态为 0200。内置命令失败返回状态 1，其他则返回 0
symlinks (+)	可以设置为多个值以控制符号链接(symlink)定位(参见 tcsh 帮助手册)
tcsh (+)	以 R.VV.PP 的格式显示的 shell 版本号，其中 R 是主版本号，VV 是当前版本，PP 是补丁号
term (+)	终端类型。通常在~/.login 文件中设置，该文件在“C/TC shell 启动”一节有详细描述
time	如果将其设为一个数字，则每当命令执行超过这个秒数后，自动执行内置命令 time。如果还有第二个词，则用作控制内置命令 time 输出的格式化字符串。下列序列可用于格式化字符串
	%c 任意的上下文切换数
	%D(非共享)数据/栈空间的平均大小，以 KB 为单位
	%E 耗用的时间(用 wall clock 命令得到)，以秒为单位
	%F 主页面故障数(需要从磁盘请求的页)
	%I 输入操作数
	%K 使用的全部空间(%X+%D)，以 KB 为单
	%k 接收到的信号数
	%M 进程可使用的最大内存数，以 KB 为单位
	%O 输出操作数
	%P 使用(%U+%S)/%E 计算的 CPU 使用率
	%R(+)次要页故障数
	%r 收到的 socket 消息数
	%S 每个 CPU 时间片进程在内核模式中执行的时间

(续表)

变　量	含　义
time(续)	%s 发出的 socket 消息数
	%U 每个 CPU 时间片进程在内核模式中执行的时间
	%W 进程换入换出的次数
	%w 非任意上下文切换(等待)数
	%X (共享)代码空间的平均大小，以 KB 为单位
	在不支持 BSD 资源限制功能的系统上，仅能使用%U、%S、%F 和%P 序列。支持资源使用报告的系统默认时间格式为%Uu %Ss %E %P %X+%Dk %I+%Oio %Fpf+%Ww，不支持资源使用报告的系统默认时间格式为%Uu %Ss %E %P
	在后来的 DYNIX/ptx 系统中，%X、%D、%K、%r 和%s 不再可用，但同时增加了下面的序列
	%Y 执行的系统调用数
	%Z 请求清零的页数
	%i 进程驻留集大小被内核增大的次数
	%d 进程驻留集大小被内核减小的次数
	%l 执行读系统调用的次数
	%m 执行写系统调用的次数
	%p 从磁盘设备读的次数
	%q 向磁盘设备写的次数
	默认时间格式为%Uu %Ss %E %P %I+%Oio %Fpf+%Ww。注意在多处理机系统上 CPU 使用率可能高于 100%
tperiod (+)	执行 periodic 特殊别名的周期，以分钟数表示
tty (+)	tty 名，如果没有与任何终端相连则为空
uid (+)	用户的实际 ID
user (+)	用户登录名
verbose	如果设置，则历史替换(如果有的话)后，每个命令都被打印
version (+)	版本 ID 标志。包含 shell 版本号(参见 tcsh)、起源、发布日期、销售商、操作系统等

9.20.2　TC shell 命令行开关

TC shell 可以使用许多命令行开关(也称为标志参数)来控制或改变它的行为。表 9-27 列出了所有的命令行开关。



表9-27　TC shell命令行开关

开　关	含　义
-	指出该shell是一个登录shell
-b	从选项处理中强退。它后面的所有shell参数将不会被当作选项。其余的参数将不被解释为shell选项。如果shell曾经进行过set-user id的设置，则必须包含此选项
-c	如果-c选项后有一个参数，则将从这个参数(文件名)中读取命令。其余的参数放入shell变量argv中
-d	shell从~/.cshdirs中加载目录栈
-Dname[=value]	将环境变量name设置为value
-e	如果被调用的命令非正常结束或返回一个非0的退出状态值，则shell将退出
-f	当启动新的TC shell时，shell忽略文件~/.tcshrc，从而快速启动
-F	shell使用fork(2)代替vfork(2)派生进程(仅用于Convex/OS)
-i	该shell是交互式的，即使它看起来并不像一个终端，但它提示用户进行输入。当输入与输出均与终端相连时，该选项是不必要的
-l	如果-l是指定的唯一参数，则该shell是登录shell
-m	即使~/.tcshrc文件不属于当前用户，shell还是加载该文件
-n	用于调试脚本。shell分析命令但并不执行它们
-q	shell接受SIGQUIT信号并能像在调试器中一样进行相应的处理。禁止作业控制
-s	从标准输入接收命令输入
-t	shell 读取一行输入并执行。反斜杠(\)可以用于转义行尾的换行符，并将该行延伸到下一行
-v	设置shell变量verbose，因此历史替换后将会回显命令输入。用于调试shell脚本
-x	设置shell变量echo，因此在历史替换和变量替换之后，命令执行之前将回显命令。用于调试shell脚本
-V	在执行~/.tcshrc文件前设置shell变量verbose
-X	在执行~/.tcshrc文件前设置shell变量echo

习题19：TC shell入门

1. init进程的作用是什么？
2. login进程有哪些作用？
3. 如何知道您所使用的shell类型？
4. 如何改变登录shell？
5. 请解释.tcshrc，.cshrc和.login文件的区别，哪一个最先执行？
6. 按如下方式编辑您的.tcshrc或.cshrc文件。
   a. 创建3个别名。
   b. 用主机名、时间和用户名重置提示符。
   c. 设置下述变量并在每个变量后加一条注释以说明该变量的作用：noclobber，history，ignoreof，savehist，prompt。

7. 键入下列命令：

```
source .tcshrc 或 source .cshrc
```

source 命令将执行什么操作？

8. 按如下方式编辑.login 文件。
   a. 欢迎用户。
   b. 若主目录不存在，则在路径名中增加主目录。
   c. 对.login 文件执行 source 命令。
9. path 与 PATH 的区别是什么？

习题 20：历史命令

1. 注销后历史事件存在哪个文件中？所显示的历史事件数量是由哪个变量控制的？savehist 变量的作用是什么？
2. 以逆序打印历史列表。
3. 不带行编号打印历史列表。
4. 键入如下命令：
   a. ls –a
   b. date '+%T'
   c. cal 2004
   d. cat /etc/passwd
   e. cd
5. 键入 history，输出是什么？
   a. 如何重新执行最后一条命令？
   b. 键入 echo a b c
   使用 history 命令重新执行仅带-c 选项的 echo 命令？
6. 使用历史命令打印并执行历史列表中以字母 d 开头的最近一条命令。
7. 执行以字母 c 开头的最近一条命令。
8. 以前一条命令的最后一个参数作为参数执行 echo 命令。
9. 使用历史替换命令将 date 命令中的 T 替换为 H。
10. 如何使用 bindkey 命令启动 vi 编辑器进行命令行编辑？
11. 如何列出编辑器命令和它们的作用？
12. 怎样知道编辑键是如何绑定的？
13. 描述变量 fignore 的作用。

习题 21：shell 元字符

1. 在提示符处键入如下内容：

```
touch ab abc a1 a2 a3 all a12 ba ba.1 ba.2 filex filey AbC ABC ABc2 abc
```

写出能执行下列操作的命令并进行测试：

a. 列出所有以 a 开头的文件。



b. 列出所有至少以一位数字结尾的文件。

c. 列出所有不以 a 或 A 开头的文件。

d. 列出所有以句点加一个数字结尾的文件。

e. 列出所有仅包含两个字母的文件。

f. 列出所有为 3 个大写字母的文件。

g. 列出所有以 11 或 12 结尾的文件。

h. 列出所有以 x 或 y 结尾的文件。

i. 列出所有以一个数字、一个大写字母或一个小写字母结尾的文件。

j. 列出所有包含一个 b 的文件。

k. 删除以 a 开头的双字符文件。

习题 22：重定向

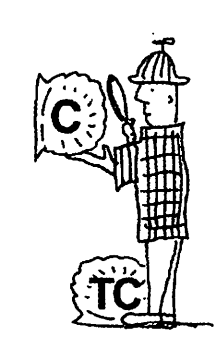
1. 与终端相连的 3 个文件流是什么？
2. 什么是文件描述符？
3. 使用什么命令可以：
   a. 将 ls 命令的输出重定向到一个 lsfile 文件？
   b. 将 date 命令的输出重定向并追加到 lsfile 文件？
   c. 将 who 命令的输出重定向到 lsfile 文件？结果如何？
   d. 单独键入 cp 结果如何？
   e. 如何将上述例子的错误信息保存到一个文件？
   f. 使用 find 命令从父目录开始搜索所有的目录文件，将标准输出存放在文件 found 中，将所有的错误存放在 found.errs 中。
   g. 什么是 noclobber？如何撤消它？
   h. 将 3 个命令的输出重定向到文件 gottemall 中。
   i. 使用管道连接 ps 命令和 wc 命令，统计出当前有多少进程在运行。

习题 23：变量与数组

1. 局部变量与环境变量有何区别？
2. 如何列出所有的局部变量？环境变量？
3. 在哪个初始化文件中存储局部变量？为什么？
4. 创建一个名为 fruit 的数组，数组里放入 5 种类型的水果。
   a. 打印数组。
   b. 打印数组的最后一个元素。
   c. 打印出数组元素的个数。
   d. 删除数组的第一个元素。
   e. 在数组中存储一个非水果类型的元素，可以吗？
5. 描述词表与字符串的区别。

chapter

10



C shell 与 TC shell 编程

10.1 简介

创建 shell 脚本的步骤

shell 脚本通常是在编辑器中编写。脚本由命令和散布其间的注释组成。注释是写在#号(#)后面，为要执行的操作提供注解。

第一行 脚本的第一行位于左上角，以#!开头(#!常常被称作 shbang)，将指明用哪个程序来执行脚本中的行。这一行通常写作：

```
#! /bin/csh 或 #!/bin/tcsh
```

#!也被称为幻数，内核根据它来确定该用哪个程序来解释脚本中的行。程序被调入内存后，内核会检查它的第一行。如果这一行是二进制数据，内核就把它作为编译生成的程序来执行。如果第一行包含#!，内核就会检查#!后面的路径，并启动该程序作为解释器。如果给出的路径是/bin/csh，就由 C shell 来解释程序中的行。如果给出的路径是/bin/tcsh，则由 TC shell 来解释程序中的行。这一行必须在脚本的最顶端，否则就会被当成注释行。

启动脚本时，C shell(TC shell)先读入并执行.cshrc(.tcshrc)文件，于是，该文件中设置的所有对象都将成为脚本的一部分。可以通过 C shell 的-f(fast，快捷)选项来阻止它将.cshrc(.tcshrc)读入您的脚本。这个选项的用法是：

```
#! /bin/csh -f 或 #! /bin/tcsh -f
```

注意，在下面所有的例子中，如果使用 tcsh，则将 shbang 行改为#!/bin/tcsh –f。

注释 注释是跟在#号(#)后的行。注释用于为脚本提供注解。如果没有注释，有时很难理解脚本究竟可以用来做什么。注释很重要，但是脚本中却经常缺少注释，甚至根本就没有注释。我们要尽量养成写注释的习惯，不光为别人着想，也方便自己。因为过了两天您可能就无法清晰地记起当时要做的是什么。



使脚本可执行 当您创建文件时，通常不会授予它执行权限。而如果要运行脚本，则必须给它执行权限。可以用 chmod 命令来打开脚本的执行权限。

范例 10-1

```
1   % chmod +x myscript
2   % ls -lF  myscript
    -rwxr--xr--x   1  ellie  0 Jul  13:00 myscript*
```

说明

1. 用 chmod 命令打开文件属主、属组和其他用户的执行权限。

2. ls 命令的输出表明所有用户都拥有文件 myscript 的执行权限。文件名尾部的星号(使用-F 选项的结果)也说明这是一个可执行程序。

编写脚本的会话实例 下面这个例子中，用户将在编辑器中创建一个脚本。用户保存脚本文件后，用 chmod 命令打开脚本的执行权限，然后执行该脚本。如果程序中有任何错误，C shell 会立即做出响应。

范例 10-2

```
(脚本: info)
    #!/bin/csh -f
    # Scriptname: info
1   echo Hello ${LOGNAME}!
2   echo The hour is `date +%H`
3   echo "This machine is `uname -n`"
4   echo The calendar for this month is
5   cal
6   echo The processes you are running are:
7   ps -ef | grep  "^ *$LOGNAME"
8   echo "Thanks for coming. See you soon\!\!"

(命令行)
    % chmod +x info
    % info
1   Hello ellie!
2   The hour is 09
3   This machine is jody
4   The calendar for this month is
5      July 2004
     S   M  Tu   W   Th  F   S
                     1   2   3
     4   5   6   7   8   9   10
    11  12  13  14  15  16  17
    18  19  20  21  22  23  24
    25  26  27  28  29  30  31
7   The processes you are running are:
    < output of ps prints here >
8   Thanks for coming. See you soon!!
```

说明

1. 问候用户。变量 LOGNAME 保存了用户的名字。BSD 系统则用变量 USER 保存用户名。花括号的作用是把变量和感叹号隔开。不需要转义这个感叹号，因为感叹号后面至少要有一个字符才会被解释为历史字符。

2. date 命令被括在反引号中。shell 将执行命令替换，并将 date 命令的输出，即当前的钟点，替换到 echo 的字符串中。

3. 命令 uname -n 显示的是机器名。

4、5. 没有把 cal 命令括在反引号中的原因是：shell 执行命令替换时，会删除输出中的所有换行符。这样就会产生一个怪模怪样的日历。而让 cal 命令独占一行就能保留其输出结果的格式。第 5 行打印了当月的日历。

6、7. 打印该用户启动的进程。BSD 系统上使用的命令则是 ps -aux，Linux 使用的是 ps au。

8. 打印该字符串。注意，末尾两个感叹号前面都加了反斜杠，这是必要的，目的是制止历史替换。[①]

10.2 读取用户输入

10.2.1 变量$<

要让脚本以交互方式运行，就要用 C shell 的一个专用变量来把标准输入读到某个变量中。符号$<可用于从标准输入读入一行，读到换行符为止，但不包括换行符，然后将该行赋给一个变量[②]。

范例 10-3

```
(脚本: greeting)
   #/bin/csh -f
   # The greeting script
1  echo -n  "What is your name? "
2  set name = $<
3  echo Greetings to you, $name.

(命令行)
   % chmod +x greeting
   % greeting
1  What is your name? Dan Savage
3  Greetings to you, Dan Savage.
```

说明

1. 将字符串回显到屏幕上。选项-n 让 echo 命令不要打印串尾的换行符。运行某些版

---

① Linux 支持很多类型的选项，包括 UNIX98、BSD 和 GNU 系统中支持的，具体请参考 ps 的帮助信息。

② 还有一种方法也可用于读取一行输入：setvariable = 'head – 1'。



本的 echo 时，可以在字符串末尾加上\c 来取消换行符，例如：echo hello\c。

2. 不管用户从终端键入什么，换行符之前的内容都作为字符串保存到变量 name 中。

3. 执行完变量替换后，打印字符串。

10.2.2 根据输入的字符串创建词表

将变量$<中的输入保存为一个字符串时，您可能会想把这个字符串分解成一个个单词来保存。

范例 10-4

```
1   % echo What is your full name\?
2   % set name = $<
    Lola Justin Lue
3   % echo Hi $name[1]
    Hi Lola Justin Lue
4   % echo $name[2]
    Subscript out of range.
5   % set name = ( $name )
6   % echo Hi $name[1]
    Hi Lola
7   % echo $name[2] $name[3]
    Justin Lue
```

说明

1. 提示用户输入。

2. 专用变量$<用字符串格式接收用户的输入。

3. 用户输入的内容 Lola Justin Lue 被保存为一个字符串，所以下标[1]就显示出整个串。数组下标从 1 开始编号。

4. 字符串 name 只有一个词，没有第 2 个词，所以使用下标[2]时，shell 提示 Subscript is out of range."，意思是“下标越界”。

5. 要用圆括号把字符串括起来创建词表。这条命令创建了一个数组。字符串被分解成一列词并且赋给变量 name。

6. 打印数组 name 的第 1 个元素。

7. 打印数组 name 的第 2 和第 3 个元素。

10.3 算术运算

脚本中的确不需要解决数学问题，但算术运算有时却是必要的，例如增减循环计数器。C shell 只支持整数的算术运算。符号@用于将计算结果赋给数值变量。

10.3.1 算术运算符

表 10-1 中的运算符用于对整数进行算术运算。这组运算符跟 C 编程语言的运算符完全

一样。请参见 10.6.2 节中表 10-6 给出的运算符优先级。C shell 还添加了 C 语言中使用的递增、递减等运算符，参见表 10-2。

表 10-1 整数运算符

运 算 符	功 能
+	加法
-	减法
/	除法
*	乘法
%	取模
<<	左移
>>	右移

表 10-2 快捷运算符

运 算 符	示 例	等 同 于
+=	@ num += 2	@ num = $num + 2
-=	@ num -= 4	@ num = $num – 4
*=	@ num *= 3	@ num = $num * 3
/=	@ num /=2	@ num = $num / 2
++	@ num++	@ num = $num + 1
--	@ num--	@ num = $num – 1

范例 10-5

```
1   % @ sum = 4 + 6
    echo $sum
    10
2   % @ sum++
    echo $sum
    11
3   % @ sum += 3
    echo $sum
    14
4   % @ sum--
    echo $sum
    13
5   % @ n = 3+4
    @: Badly formed number
```

说明

1. 把 4 加 6 的结果赋给变量 sum(@后的空格是必需的)。
2. 将变量 sum 加 1。
3. 将变量 sum 加 3。
4. 将变量 sum 减 1。
5. 符号@后面、运算符两侧都必须有空格。



10.3.2 浮点算术运算

C shell 不支持浮点算术运算，因此，如果您需要执行更复杂的数学运算，可以使用 UNIX 的实用程序。

如果您需要执行复杂的计算，bc 和 nawk 这两个实用程序很有用。

范例 10-6

```
(命令行)
1  set n='echo "scale=3; 13 /2 " | bc'
   echo $n
   6.500
2  set product = 'nawk -v x=2.45 -v y=3.124 'BEGIN{\
   printf "%.2f\n",x*y}''
   % echo $product
    7.65
```

说明

1. echo 命令的输出通过管道传给 bc 程序。精度(scale)被设为 3，精度指的是保留到小数点后第几位。这个例子执行的计算是用 2 除 13。整个管道被括在反引号中，shell 对其执行命令替换，将结果赋给变量 n。

2. nawk 程序从由命令行传入的参数列表中取得变量的值。传给 nawk 的每个参数前面都有一个-v 开关，例如-v x=2.45 和-v y=3.124。两个数相乘后，printf 函数以保留小数点后两位的格式打印结果。输出结果被赋给变量 product。

10.4 脚本调试

C/TC shell 脚本常常因为一些简单的语法或逻辑错误而不能正确运行。csh 命令提供了一些选项来帮助您调试脚本。参见表 10-3。

表 10-3 回显(-x)与详细(-v)

选　项	含　义
csh 和 tcsh 的参数	
csh –x scriptname tcsh –x scriptname	对脚本的每一行执行变量替换，然后显示并执行该行
csh –v scriptname tcsh –v scriptname	先按输入时的原样显示脚本的每一行，然后执行它
csh –n scriptname tcsh –n scriptname	解释命令，但不执行命令
作为 set 命令的参数	
set echo	对脚本的每一行执行变量替换，然后显示并执行该行
set verbose	先按输入时的原样显示脚本的每一行，然后执行
作为脚本的第一行	
#!/bin/csh –xv #!/bin/tcsh –xv	同时打开回显和详细功能。这两个选项可以单独使用，也可以与其他的 csh 参数结合使用

范例 10-7

```
(-v 选项和-x 选项)
1   % cat practice
    #!/bin/csh
    echo Hello $LOGNAME
    echo The date is `date`
    echo Your home shell is $SHELL
    echo Good-bye $LOGNAME

2   % csh -v practice
    echo Hello $LOGNAME
    Hello ellie
    echo The date is `date`
    The date is Sun May 23 12:24:07 PDT  2004
    echo Your login shell is $SHELL
    Your login shell is /bin/csh
    echo Good-bye $LOGNAME
    Good-bye ellie

3   % csh -x practice
    echo Hello ellie
    Hello ellie
    echo The date is `date`
    date
    The date is Sun May 23 12:24:15 PDT  2004
    echo Your login shell is /bin/csh
    Your login shell is /bin/csh
    echo Good-bye ellie
    Good-bye ellie
```

说明

1. 显示一个 C shell 脚本的内容。该脚本中都是些变量替换和命令替换的命令。通过此例可以使我们明白回显(echo)和详细(verbose)这两个选项之间的区别。

2. csh 的-v 选项激活详细功能。脚本的每一行先被原样显示，然后执行。

3. csh 的-x 选项激活回显功能。执行变量替换和命令替换后，shell 显示脚本中每一行，然后执行它。使用这一功能可以检查命令和变量替换的结果中顶替了哪些部分，所以它比详细选项更为常用。

范例 10-8

```
(Echo 和 Verbose)
1   % cat practice
    #!/bin/csh
    echo Hello $LOGNAME
    echo The date is `date`
    set echo
    echo Your home shell is $SHELL
    unset echo
    echo Good-bye $LOGNAME
```




```
    % chmod +x practice

2   % practice
    Hello ellie
    The date is Sun May 26 12:25:16 PDT  2004
--> echo Your login shell is /bin/csh
--> Your login shell is /bin/csh
--> unset echo
    Good-bye ellie
```

说明

1. 在脚本内设置并清除回显选项。如果你在脚本的某个部分遇到瓶颈，可以用这种方式来调试，而不必回显整个脚本的每一行。

2. -->标出打开回显选项的部分。每一行都在变量替换和命令替换完成后被打印，然后才被执行。

范例 10-9

```
1   % cat practice
    #!/bin/csh
    echo Hello $LOGNAME
    echo The date is `date`
    set verbose
    echo Your home shell is $SHELL
    unset verbose
    echo Good-bye $LOGNAME

2   % practice
    Hello ellie
    The date is Sun May 23 12:30:09 PDT  2004
--> echo Your login shell is $SHELL
--> Your login shell is /bin/csh
--> unset verbose
    Good-bye ellie
```

说明

1. 在脚本内设置并清除详细选项。

2. -->标出打开详细选项的部分。按照输入的原样打印出每一行，然后执行它。

10.5　命令行参数

shell 脚本可以接受命令行参数。参数将对程序的行为产生某种影响。C shell 把命令行参数赋值给位置参量，而对位置参量的个数不加任何特殊限制(而 Bourne shell 限定只能使用 9 个位置参量)。位置参量属于数值变量。脚本名被赋给$0，其后的所有词则被依次赋给$1、$2、$3……${10}、${11}等参量。$1 是第一个命令行参数。除了使用位置参量，C shell 还提供了一个内置数组 argv。

位置参量和 argv

使用 argv 数组时，必须提供合法的下标来对应命令行传入的参数，否则，C shell 就会发出报错信息“Subscript out of range”(下标越界)。argv 数组不包括脚本名。第一个参数是 $argv[1]，$#argv 则代表参数的个数($#argv 是参数个数的唯一表示方法)。请参见表 10-4 中的命令行参数列表。

表 10-4　命令行参数

参　数	含　义
$0	脚本名
$1,$2, …, ${10}…	头两个位置参量的引用方式是在数字前加一个美元符。打印第 10 个位置参量时，数字 10 用花括号括起来，这样就避免打印成第一个参量后跟一个数字 0
$*	所有位置参量
$argv[0]	无效，不会打印任何内容。C shell 的数组下标从 1 开始
$argv[1] $argv[2] …	第 1 个、第 2 个、等位置参量
$argv[*]	所有参数
$argv	所有参数
$#argv	参数的个数
$argv[$#argv]	最后一个参数

范例 10-10

(脚本)

```
#!/bin/csh -f
# The greetings script
# This script greets a user whose name is typed in at the command line.
1  echo $0 to you $1 $2 $3
2  echo Welcome to this day `date | awk '{print $1, $2, $3}'`
3  echo Hope you have a nice day, $argv[1]\!
4  echo Good-bye $argv[1] $argv[2] $argv[3]
```

(命令行)

```
   % chmod +x greetings
   % greetings Guy Quigley
1  greetings to you Guy Quigley
2  Welcome to this day Fri Aug 28
3  Hope you have a nice day, Guy!
4  Subscript out of range
```

说明

1. 将显示脚本的名称和头 3 个位置参量。来自命令行的参数只有两个，即 Guy 和 Quigley，所以$1 变成了 Guy，$2 变成了 Quigley，$3 则未定义。

2. 用单引号引用 awk 命令，这样，shell 就不会把 awk 的字段号$1、$2 和$3 误认成位置参量(不要把 awk 的字段指示符$1、$2 和$3 跟 shell 的位置参数混淆)。



3. shell把来自命令行的值赋给argv数组。Guy被赋给argv[1]，然后显示其值。在脚本中，可以用argv数组，也可以用位置参量来指代命令行参数。这两种方法的区别在于引用未赋值的位置参量不会报错，而引用未赋值的argv数组元素则会导致脚本发出报错消息"Subscript out of range"(下标越界)并退出。

4. 因为argv[3]没有赋值，所以shell打印报错消息"Subscript out of range"。

10.6 条件结构与流控制

在脚本中作判断时，可以使用if、if/else、if/else if和switch这些命令。这些命令基于表达式的真假作出判断，由此来控制程序执行的方向。

10.6.1 测试表达式

以运算符分隔的一组操作数组成了表达式。表10-5和表10-6分别列出了各种运算符及它们的优先关系。如果要测试表达式，就要用圆括号把它括起来。C shell计算表达式后将返回一个结果为0或非0的数值。结果非0，则表达式为真。结果为0，则表达式为假。

表10-5 比较和逻辑运算符

运 算 符	含 义	示 例
==	等于	$x == $y
!=	不等于	$x != $y
>	大于	$x > $y
>=	大于或等于	$x >= $y
<	小于	$x < $y
<=	小于或等于	$x <= $y
=~	字符串匹配	$ans =~ [Yy]*
!~	字符串不匹配	$ans !~ [Yy]*
!	逻辑非	! $x
||	逻辑或	$x || $y
&&	逻辑与	$x && $y

shell对逻辑与(&&)表达式求值的顺序是从左往右。如果第一个表达式(&&前的表达式)为假，shell就把整个表达式的结果定为假，不再检查余下的表达式。只有当逻辑与运算符(&&)两边的表达式都为真时，整个表达式才为真。

```
( 5 && 6 )    Expression is true
( 5 && 0 )    Expression is false
( 0 && 3 && 2 )   Expression is false
```

表 10-6 运算符的优先级

运 算 符	优 先 级	含 义
()	高	改变优先级，用于结合
~	↓	求反
!	↓	逻辑非，否定
* / %	↓	乘，除，求模
+ -	↓	加，减
<< >>	↓	左移位和右移位
> >= < <=	↓	关系运算符：大于，小于
== !=	↓	相等关系：等于，不等于
=~ !~	↓	模式匹配：匹配，不匹配
&	↓	按位与
^	↓	按位异或
|	↓	按位或
&&	↓	逻辑与
||	低	逻辑或

对逻辑或(||)表达式求值时，如果运算符||左边的第一个表达式为真，shell 就将整个表达式的值设为真，而不再检查余下的其他表达式。只要其中有一个表达式为真，整个逻辑或表达式的值就为真。

```
( 5 || 6 )   Expression is true
( 5 || 0 )   Expression is true
( 0 || 0 )   Expression is false
( 0 || 4 || 2 )   Expression is true
```

逻辑非是一元运算符。也就是说，它只对单个表达式求值。如果逻辑非运算符右边的表达式为真，则整个表达式的值为假。若为假，则整个表达式的值为真。

```
! 5  Expression is false
! 0  Expression is true
```

10.6.2 优先级和组合规则

测试表达式时，C shell 和 C 语言一样会用到优先级和组合规则。如果有一个像下面这样混合使用多种运算符的表达式：

```
@ x = 5 + 3 * 2
echo $x
11
```

shell 将按某个特定顺序来读取运算符。优先级(precedence)指的是运算符执行的先后次



序。组合规则是指当优先级相等时，shell 读表达式的方向是从左向右还是从右向左[③]。组合的次序将与算术表达式的不同(shell 脚本很少用到算术表达式)，它是从左向右进行的。可以用圆括号来改变组合次序(参见表 10-6)。

```
@ x =  (5 + 3)* 2
echo $x
16
```

表达式可以分为数值表达式、关系表达式和逻辑表达式 3 类。数值表达式使用下面这些算术运算符：

```
+ - * / ++ -- %
```

关系表达式使用下面这些运算符，它们产生的结果无外乎真(非 0)或假(0)：

```
> < >= < <= == !=
```

逻辑表达式使用的运算符是：

```
! && ||
```

10.6.3 if 语句

形式最简单的条件语句是 if 语句。If 表达式经测试后，如果值为真，shell 就执行关键字 then 后的语句直至遇到 endif 为止。关键字 endif 用于终结 if 语句块。if 语句可以嵌套，但是必须保证每一个 if 语句都有对应的 endif 来终结。endif 对应的是前面最近一个未闭合的 if 语句。

格式

```
if (表达式) then
  命令
  命令
endif
```

范例 10-11

(脚本：检查参数)

```
1   if ( $#argv != 1 ) then
2       echo "$0 requires an argument"
3       exit 1
4   endif
```

说明

1. 如果从命令行传入的参数的个数($#argv)不等于 1，则执行 then 后面的语句。
2. 如果第一行为真，则执行此行和第 3 行。
3. 程序以值 1 退出，表示运行失败。
4. 每一个 if 块都要用 endif 语句结束。

---

③ 优先级相同时，算术表达式从右向左进行组合。

10.6.4 测试未设置或值为空的变量

专用变量$?用于测试某个变量是否已被设置。变量被设置(包括设为空值)时，$?变量返回真。

范例 10-12

```
(摘自文件.cshrc 或.tcshrc)
if ( $?prompt ) then
   set history = 32
endif
```

说明

每次启动一个新的 csh(或 tcsh)程序时，都要执行.cshrc(或.tcshrc)文件。$?可以用来测试某个变量是否已被设置。这个例子中，shell 检查是否已设置提示符。如果设置了提示符，说明正在运行的是一个交互式的 shell，而不是脚本。只有在交互式使用时才需要设置提示符。命令历史机制只在交互式使用 shell 时才有用，因此，如果您正在运行脚本，shell 就不会设置命令历史。

范例 10-13

```
(脚本)
    echo -n "What is your name? "
1   set name = $<
2   if ( "$name" != "" ) then
       grep "$name" datafile
    endif
```

说明

1. 提示用户输入。即使用户直接按下回车键，变量也会被设置，只不过被设为空值。

2. 变量被双引号括着，这样，即使用户输入的 name 值不止一个词，表达式依然可以取值。如果把这对引号删掉，而用户的输入中既有姓又有名，shell 就会报错并退出，返回的错误信息将是："if: Expression syntax."。空的双引号代表空字符串。

10.6.5 if/else 语句

if/else 结构是一个双路分支控制结构。如果 if 命令后的表达式为真，shell 将执行 if 后面的语句块。否则就执行 else 后面的语句块。endif 对应最内层的 if 语句并终结该语句。

格式

```
if (表达式) then
  命令
else
  命令
endif
```

范例 10-14

```
1   if ( $answer =~ [Yy]* ) then
2        mail bob < message
```



```
3   else
4       mail john < datafil
5   endif
```

说明

1. 此行的含义是：如果$answer 的值符合条件“Y 或 y 后面跟零个或多个字符”，则执行第 2 行。否则，执行第 3 行(星号*是 shell 通配符)。

2. 通过邮件将文件 message 的内容发给用户 bob。

3. 如果第一行不为真，执行 else 下面的行。

4. 通过邮件将文件 datafile 的内容发给用户 john。

5. endif 终结 if 语句块。

10.6.6 逻辑表达式

逻辑运算符为&&、| | 和!。与运算符对&&左侧的表达式求值。如果为真，则测试&&右侧表达式且必须为真。如果有一个表达式为假，则整个表达式为假。或运算符对| |左侧的表达式求值，如果为真，则整个表达式为真。如果为假，则测试| |右侧的表达式，如果为真，则整个表达式为真。如果左右两侧的表达式都为假，则整个表达式为假。非运算符是一元运算符，它对！右侧表达式求值，如果为真则它为假，如果为假则它为真。

C/TC shell 的-x 选项(称作回显选项)使您能够追踪脚本执行时究竟做了哪些工作(参见第 15 章“调试 shell 脚本”)。

范例 10-15

(脚本：使用逻辑表达式并检测值)

```
    #!/bin/csh -f
    # Scriptname: logical
    set x = 1
    set y = 2
    set z = 3
1   if ( ( "$x" && "$y" ) || ! "$z" ) then
        # Note: grouping and parentheses
2       echo TRUE
    else
        echo FALSE
    endif
```

(输出)

```
3   % csh -x logical
    set x = 1
    set y = 2
    set z = 3
    if ( ( 1 && 2 ) || ! 3 ) then
    echo TRUE
    TRUE
    else
    %
```

说明

1. 计算这个逻辑表达式。第一个表达式被括在圆括号中(并非必要，因为&&的优先级本来就高于||)。嵌套使用时，括号间不需要加空格，但是逻辑非运算符(!)后面必须有一个空格。

2. 如果上一行表达式的值为真，那么执行这一行。

3. 用-x 开关执行 csh 程序，这样就激活了回显功能。shell 对脚本的每一行执行变量替换，然后回显出来。

10.6.7 if 语句和单条命令

如果表达式后面只有一条单独的命令，那么就不需要使用关键字 then 和 endif。

格式

```
if (表达式) 单条命令
```

范例 10-16

```
if ($#argv == 0) exit 1
```

说明

测试表达式，如果命令行参数的个数$#argv 等于 0，程序就以状态 1 退出。

10.6.8 if/else if 语句

if/else if 结构提供了一种多路判断机制。该结构能测试多个表达式，如果某个表达式为真，就执行它后面的语句块。如果所有表达式均不为真，则执行 else 块。

格式

```
if (表达式) then
  命令
  命令
else if (表达式) then
  命令
  命令
else
  命令
endif
```

范例 10-17

```
(脚本名：grade)
    #!/bin/csh -f
    # Scriptname: grade
    echo -n "What was your grade? "
    set grade = $<
1   if ( $grade >= 90 && $grade <= 100 ) then
        echo "You got an A\!"
2   else if ( $grade > 79 ) then
        echo "You got a B"
3   else if ( $grade > 69 ) then
```




```
        echo "You're average"
    else
4       echo "Better study"
5   endif
```

说明

1. 如果 grade 大于或等于 90，并且小于或等于 100，则打印“You got an A!”。&&两侧的表达式必须同时为真，否则，程序控制将转向第 2 行的 else if。

2. 如果行 1 为假，就测试行 2 的表达式，若为真，则打印“You got a B”。

3. 如果行 1 和行 2 均为假，再试试这行。如果这个表达式为真，则打印“You are average”。

4. 如果上面 3 个表达式的测试结果皆为假，则执行 else 块中的语句。

5. endif 结束整个 if 结构。

10.6.9 退出状态和变量 status

每条 UNIX/Linux 命令都会返回一个退出状态。如果执行成功，命令返回退出状态 0；如果执行失败，命令返回非 0 的退出状态。你可以通过查看 C shell 变量 status 的值来判断命令是否执行成功。变量 status 将保存 shell 执行的上一条命令的退出状态。

范例 10-18

```
1   % grep ellie /etc/passwd
    ellie:pHAZk66gA:9496:41:Ellie:/home/jody/ellie:/bin/csh
2   % echo $status
    0                # Zero shows that grep was a success

3   % grep joe /etc/passwd
4   % echo $status
    1                # Nonzero shows that grep failed
```

说明

1. grep 程序在文件/etc/passwd 中找到 ellie。
2. 如果找到模式 ellie，grep 程序就会在退出时返回状态 0。
3. grep 程序未能在文件/etc/passwd 中找到 joe。
4. 未能找到指定模式时，grep 程序返回非 0 的状态值 1。

10.6.10 从 shell 脚本中退出

使用 shell 脚本里的 exit 命令会将退回到 shell 提示符界面。exit 命令可以带一个整数值作为参数来说明退出类型。非 0 的参数表示失败，参数 0 则表示成功。这个参数的值必须在 0~255 之间。

范例 10-19

```
(检查 Shell 脚本)
    #!/bin/csh -f
1   if ( $#argv != 1 ) then
```

```
2         echo "$0 requires an argument"
3         exit 2
4   endif
```

（命令行）

```
5   % checkon
    checkon requires an argument
6   %echo $status
    2
```

说明

1. 如果由命令行传入的参数的个数($#argv)不等于 1，就转到第 2 行。
2. echo 命令打印出脚本名($0)和字符串“requires an argument”。
3. 程序退回到提示符，退出状态的值为 2。这个值将被保存在父 shell 的变量 status 中。
4. 条件结构 if 的结尾。
5. 在命令行上执行程序 checkon，未指定参数。
6. 程序以状态值 2 退出，状态值被保存在变量 status 中。

10.6.11 使用别名创建错误信息

在出现某种错误后从脚本退出时，可以创建自定义的错误诊断信息。这可以通过别名来实现，参见范例 10-20。

范例 10-20

（脚本）

```
   #!/bin/tcsh -f
   # Scriptname: filecheck
   # Usage: filecheck filename
1  alias Usage 'echo "   Usage: $0 filename\!*" ; exit 1'
2  alias Error 'echo "   Error: \!* "; exit 2'
3  set file=$1
4  if ( $#argv == 0 ) then
     Usage
   endif
5  if ( ! -e $file ) then
       Error "$file does not exist"
   endif
```

（命令行）

```
% filecheck
Usage: filecheck filename
% echo $status
1

% filecheck xyzfile
Error: xyzfile does not exist
% echo $status
2
```



10.6.12 在脚本中使用变量 status

变量 status 也可以被脚本用来测试命令的退出状态。变量 status 的值是上一条被执行的命令的退出状态。

范例 10-21

(脚本)

```
    #!/bin/csh -f
1   ypmatch  $1  passwd >& /dev/null
2   if ( $status == 0 ) then
3          echo Found $1 in the NIS database
    endif
```

说明

1. ypmatch 程序检查 NIS 数据库，看看作为第一个参数传入的用户名是否在这个数据库中。
2. 如果上一条命令返回的 status 的值是 0，就执行 then 语句块。
3. 如果对 if 测试表达式求值的结果为真，就执行该行。

10.6.13 在条件结构中对命令求值

C shell 可以在条件语句中计算表达式。如果要在条件表达式中对命令求值，就必须用花括号把命令括起来。如果命令执行成功，也就是说命令返回的退出状态为 0，花括号就会指示 shell 将该表达式的值记为真(1)[④]。命令运行失败时，其退出状态不为 0，则表达式的值为假(0)。

在条件结构中使用命令时，了解命令的退出状态很重要。例如，如果发现了它要查找的模式，grep 程序返回退出状态 0；未能找到，则返回 1；如果找不到指定的文件，返回值则是 2。awk 或 sed 查找模式时，无论是否找到，程序都会返回 0。awk 和 sed 把语法正确与否作为判断是否成功的标准，就是说，只要您输入的命令是正确的，awk 和 sed 的退出状态就会是 0。

如果把感叹号放在表达式前，它将对整个表达式取反，因此，若表达式原来为真，加感叹号后就变成假，反之亦然。用感叹号取反时，必须在它后面加一个空格，否则，C shell 将调用命令历史机制。

格式

```
if { ( 命令 ) }  then
  命令
  命令
endif
```

范例 10-22

```
#!/bin/csh -f
1  if  { ( who | grep $1 >& /dev/null ) } then
```

④ shell 会转换命令的退出状态，因此，表达式将返回结果真或假。

```
2       echo $1 is logged on and running:
3       ps -ef | grep "^ *$1"      # PS -aux for BSD
4   endif
```

说明

1. 将 who 命令的输出结果经由管道发给 grep 命令。所有的输出都被发送到/dev/null，即 UNIX/Linux 的“位桶(bit bucket)”。who 命令的结果被送到 grep，grep 在 who 的结果中查找变量$1(第一个命令行参数)中保存的用户名。如果 grep 执行成功并且找到了指定的用户，就返回退出状态 0。接下来则由 shell 转化 grep 命令的退出状态，返回结果 1 或真。如果 shell 算出该表达式的值为真，就执行 then 和 endif 之间的命令。

2. 如果 C/TC shell 求得第 1 行的表达式的值为真，就执行第 2 行和第 3 行。

3. 打印所有正在运行的、属于$1 的进程。

4. endif 终结 if 语句。

格式

```
if ! { (命令) } then
```

范例 10-23

```
1   if  !  { ( ypmatch $user passwd >& /dev/null ) } then
2       echo $user is not a user here.
        exit 1
3   endif
```

说明

1. 使用网络时，ypmatch 命令可以检索 NIS 的 passwd 文件。如果 ypmatch 命令在 passwd 文件中成功找到指定用户($user)，这个表达式的结果就是真。表达式前面的感叹号(！)用于对表达式求非(也称取反)；即原来为真的表达式，现在为假；原来为假则现在为真。

2. 表达式非真表示找不到指定用户，就执行此行。

3. endif 终结这个 if 块。

10.6.14 goto 命令

goto 命令可以跳到程序中某个标签处，并从该位置开始执行。尽管有很多程序员不赞成使用 goto，但需要跳出嵌套循环时，它确实是一种有效的途径。

范例 10-24

(脚本)

```
    #!/bin/csh -f
1   startover:
2   echo "What was your grade? "
    set grade = $<
3   if ( "$grade" < 0 || "$grade" > 100 ) then
4       echo "Illegal grade"
5       goto startover
    endif
    if ( $grade >= 89 ) then
```



```
        echo "A for the genius\!"
    else if ( $grade >= 79 ) then
        .. < Program continues >
```

说明

1. 标签是一个用户定义的词，这个词后面有一个冒号。本行的这个标签名为 startover。执行程序时，如果不明确指示 shell 跳转到标签处，shell 将忽略该标签。

2. 提示用户进行输入。

3. 如果表达式为真，(用户输入的成绩小于 0 或大于 100)，就打印字符串"Illegal grade"，然后用 goto 语句跳到指定的标签 startover 处。程序从该位置开始继续执行。

4. 如果测得 if 表达式为真，则打印该行。

5. goto 将控制转到第一行，从标签 startover 之后开始执行。

10.6.15 C shell 文件测试

C/TC shell 都提供了一组内置选项用于测试文件的属性，以判别指定的文件是不是目录、是不是普通文件(即不是目录)或文件是否可读之类的问题。TC shell 在这个方面增加了许多新特性，参见 10.6.18 节"TC shell 文件测试"。其他类型的文件测试则可用 UNIX/Linux 的 test 命令来执行。表 10-7 列出了用于文件测试的内置选项。

表 10-7 C shell 文件测试

测 试 标 志	测试何时为真
-d	该文件是个目录
-e	该文件已存在
-f	该文件是个普通文件
-o	该文件归当前用户所有
-r	当前用户可以读该文件
-w	当前用户可以写该文件
-x	当前用户可以执行该文件
-z	该文件长度为 0

范例 10-25

```
#!/bin/csh -f
1  if ( -e file ) then
       echo file exists
   endif
2  if ( -d file ) then
        echo file is a directory
   endif

3  if ( ! -z file ) then
        echo file is not of zero length
   endif
```

```
4   if ( -r file && -w file ) then
        echo file is readable and writable
    endif
```

说明

1. 这条语句的含义是：如果指定的文件已存在，则执行 then 后面的语句。

2. 这条语句的含义是：如果指定的文件是个目录，则执行 then 后面的语句。

3. 这条语句的含义是：如果指定的文件长度为 0，则执行 then 后面的语句。

4. 这条语句的含义是：如果指定的文件可读并且可写，则执行 then 后面的语句。可以将多个文件测试标志写在一起，这样就只需在文件名前加一个选项。例如：-rwx file(相当于：-r file && -w file && -x file)。

10.6.16 test 命令与文件测试

UNIX/Linux 的 test 命令不仅包括了 C shell 的内置选项，还提供了很多其他选项。请参见表 10-8 中列出的 test 的选项。可能需要用这些补充选项来测试一些不太常用的文件属性，例如是否为块文件、特殊字符文件或 setuid 文件等。test 命令先对表达式求值，然后返回一个退出状态，成功时返回 0，失败则返回 1。在 if 条件语句中使用 test 命令时，必须用花括号把它括起来，这样 shell 才能求出正确的退出状态[⑤]。

表 10-8 用 test 命令进行文件测试

选　项	测试返回真值的情形
-b	测试对象是一个块特殊文件
-c	测试对象是一个字符特殊文件
-d	测试对象已存在并且是一个目录文件
-f	测试对象已存在并且是一个普通文件
-g	测试对象的 set-group-ID 位被置 1
-k	测试对象的 sticky 位被置 1
-p	测试对象是一个管道文件
-r	当前用户可以读测试对象
-s	测试对象已存在并且非空
-t n	n 是终端的文件描述符
-u	测试对象的 set-user-ID 位被置 1
-w	当前用户可以写测试对象
-x	当前用户可以执行测试对象

范例 10-26

```
1   if { test -b file } echo file is a block special device file

2   if { test -u file }  echo file has the set-user-id bit set
```

⑤ 给脚本取名为 test 是一个常见的错误。因为如果在您的搜索路径中，UNIX 的 test 命令所在目录更靠前，shell 就会执行 test 命令而不是 test 脚本。这样，test 命令可能显示报错信息，也可能什么都不显示(如果语法是正确的)。



说明

1. 这条语句的含义是：如果被测文件是一个块特殊文件(通常保存在目录/dev 下)，则执行后面的语句。

2. 这条语句的含义是：如果被测文件是一个 setuid 程序(设置用户 ID 程序)，则执行后面的语句。

10.6.17 条件结构的嵌套

条件语句可以嵌套使用。但每个 if 都必须有对应的 endif(else if 则不需要)。建议将嵌套语句缩进并且相应的 if 和 endif 对齐，这样才能提高阅读和调试程序的效率。

范例 10-27

(脚本)

```
    #!/bin/csh -f
    # Scriptname: filecheck
    # Usage: filecheck filename
    set file=$1
1  ┌if (! -e $file) then
   │    echo "$file does not exist"
   │    exit 1
   └endif
2  ┌if (-d $file) then
   │    echo "$file is a directory"
3  │else if (-f $file) then
4  │    ┌if (-r $file && -x $file)then      # Nested if construct
   │    │     echo "You have read and execute permission on $file"
5  │    └endif
   │else
   │     print "$file is neither a plain file nor a directory."
6  └endif
```

(命令行)

```
$ filecheck testing
You have read and execute permission of file testing.
```

说明

1. 如果 file(执行变量替换后得到的文件名)是一个不存在的文件(注意逻辑非运算符!)，就执行关键字 then 下面的语句。退出状态 1 表示程序运行失败。

2. 如果文件是一个目录，则打印“testing is a directory”。

3. 如果文件不是目录，判断它是否是普通文件。若是，则执行下一条语句，进入最内层的 if 结构。

4. 这个 if 被嵌套在前一个 if 里。如果文件可读、可写并且可执行，则执行 then 后面的语句。这个 if 有它自己的 endif，endif 被写在对应 if 的同一列，以表明其归属。

5. 这个 endif 终结内层的 if 结构。

6. 这个 endif 终结外层的 if 结构。

10.6.18 TC shell 文件测试

与 C shell 一样，TC shell 也提供了一组内置选项用于测试文件的属性，以判别指定的文件是不是目录、是不是普通文件(即不是目录)或文件是否可读之类的问题。不同的是，TC shell 提供了另外的一些技术来进行文件测试。下面将详细介绍。

表 10-9 中所列出的操作代表用户对文件或目录属性进行的测试，如果成功，则返回 1，失败则返回 0。新增的文件查询内置选项在 10.6.20 节中的表 10-10 中列出。TC shell 可以将这两种选项组合在一起使用(而 C shell 则不允许)。-rwx 与-r&&-w&&-x 这两种形式等价。

表 10-9 TC Shell 文件测试

选 项	测试返回真值的情形
-b	测试对象是一个块特殊文件
-c	测试对象是一个字符特殊文件
-d	测试对象是一个目录
-e	文件存在
-f	测试对象是一个普通文件
-g	测试对象的 set-group-ID 位被置 1
-k	测试对象的 sticky 位置 1
-l	测试对象是一个符号链接
-L	在多操作列表中将后续操作符应用于符号链接，而非该符号链接所指向的文件
-o	当前用户拥有该文件
-p	文件是一个命名管道
-r	当前用户可以读测试对象
-R	测试对象被移动过(仅用于 convex)
-s	文件大小非 0
-S	测试对象是一个套接字特殊文件
-t file	file(必须为数字)是某个终端设备打开的文件描述符
-w	当前用户可写测试对象
-x	当前用户可执行测试对象
-z	文件大小为 0

范例 10-28

(脚本)

```
    #!/bin/tcsh -f
    # Scriptname: filetest1
1   if ( -e file ) then
      echo file exists
    endif
2   if ( -d file ) then
      echo file is a directory
    endif
3   if ( ! -z file ) then
      echo file is not of zero length
```

```
   endif

4  if ( -r file && -w file && -x file) then
     echo  file is readable and writable and executable.
   endif
5  if ( -rwx file ) then
     echo file is readable and writable and executable.
   endif
```

说明

1. 该语句含义为：如果文件存在，则执行 then 后面的语句。

2. 该语句含义为：如果文件是一个目录，则执行 then 后面的语句。

3. 该语句含义为：如果文件大小非 0，则执行 then 后面的语句。

4. 该语句含义为：如果文件可读可写可执行，则执行 then 后面的语句。可以在文件名前加上单个选项(如–r file && –w file && –x file)

5. 将文件测试标志组合在一起(仅用于 tcsh)，如-rwx file。

范例 10-29

(脚本)

```
   #!/bin/tcsh -f
   # Scriptname: filetest2
1  foreach file (`ls`)
2    if ( -rwf $file ) then
3       echo "${file}: readable/writable/plain file"
     endif
   end
```

(输出)

```
3  complete: readable/writable/plain file
   dirstack: readable/writable/plain file
   file.sc: readable/writable/plain file
   filetest: readable/writable/plain file
   glob: readable/writable/plain file
   modifiers: readable/writable/plain file
   env: readable/writable/plain file
```

说明

1. foreach 循环在 UNIX/Linux ls 程序产生的文件列表中迭代，一个文件接一个文件，将每个文件名赋值给变量 file。

2. 如果文件可读写并且是一个普通文件，则第 3 行被执行。文件测试选项只有在 tcsh 中才可以组合使用，csh 中则不行。

3. 如果被测试的文件可读写并且可执行，则本行被执行。

10.6.19 内置命令 filetest(tcsh)

tcsh 内置命令 filetest 将某个文件查询操作应用在一个或多个文件，并返回由空格隔开的一列数字，1 代表成功，0 代表失败。

范例 10-30

```
1   > filetest -rwf dirstack file.sc xxx
    1 1 0
2   > filetest -b hdd
    1
3   > filetest -lrx /dev/fd
    1
```

说明

1. 测试 dirstack、file.sc 和 xxx3 个文件是否可读、可写，且为普通文件。前两个文件均返回 1，最后一个返回 0。

2. hdd 文件是一种块设备特殊文件，因此 filetest 命令返回 1。否则，返回 0。

3. fd 文件可读、可执行并且是一个符号链接文件，因此 filetest 命令返回 1。否则返回 0。

10.6.20 新增的 TC shell 文件测试操作

表 10-10 显示了所返回的文件信息附加的文件测试操作符集合(仅用于 tcsh)。因为返回值并不是真或假，所以用-1 指示失败(F 除外，它返回“:”)。

表 10-10 新增的 TC shell 文件测试

操 作 符	作 用
-A	文件最后访问时间，其数值表示自 1970 年 1 月 1 日以来累计的秒数
-A:	与 A 类似，但使用时间戳格式，如 Fri. Aug. 27 16:36:10 2004
-M	文件最后修改时间
-M:	与 M 类似，但使用时间戳格式
-C	文件索引节点最后修改时间
-C:	与 C 类似，但使用时间戳格式
-F	组合文件标识符，以设备：索引节点的形式表示
-G	组 ID 号
-G:	组名称，如果组名称未知，则使用组 ID 号
-L	符号链接指向的文件名
-N	链接(硬链接)数
-P	八进制表示的权限(最前面不带 0)
-P:	与 P 类似，但最前面带 0
-Pmode	等价于-P file & mode；例如，假设文件对同组用户及其他用户可写，则-P22 file 将返回 22。如果仅同组用户可写，则返回 20。如果都不可写则返回 0
-Mode:	与 PMode 类似，但最前面带 0
-U	用户 ID 号
-U:	用户名，如果用户名未知，则使用用户 ID 号
-Z	以字节为单位的大小

范例 10-31

```
1   > date
```



```
   Wed Apr 22 13:36:11 PST 2004
2  > filetest -A myfile
   934407771
3  > filetest -A: myfile
   Wed Apr 22 14:42:51 2004
4  > filetest -U myfile
   501
5  > filetest -P: myfile
   0600
   > filetest -P myfile
   600
```

说明

1. 打印当前日期。
2. 使用-A 选项后，内置命令 filetest 打印 myfile 最后访问时间(以秒为单位)。
3. 使用-A: 选项后，内置命令 filetest 以时间戳格式打印 myfile 最后访问时间。
4. 使用-U 选项，内置命令 filetest 打印 myfile 文件所有者的用户 ID 号。
5. 使用-P: 选项，内置命令 filetest 以加前缀 0 的八进制形式打印文件权限，无冒号时，不加前缀 0。

范例 10-32

(脚本)

```
   #!/bin/tcsh -f
   # Scriptname: filecheck
   # Usage: filecheck filename
1  alias Usage 'echo "   Usage: $0 filename\!*" ; exit 1'
2  alias Error 'echo "   Error: \!* "; exit 2'
3  set file=$1                     # The first argument, $1, is assigned
4  if ( $#argv == 0 ) then         # to a variable called
file
      Usage
    Endif
5  if ( ! -e ifile ) then
       Error "$file does not exist"
   endif
6  if ( -d $file ) then
      echo "$file is a directory"
7  else if (-f $file) then
8    if ( -rx $file ) then       # nested if construct
        echo "You have read and execute permission on $file"
9    endif
   else
     print "$file is neither a plain file nor a directory."
10 endif
```

(命令行)

```
   % filechecli testingi
   You have read and execute permission on file testing1.
```

说明

1. 别名 Usage 可以用于产生一个错误信息并退出程序。
2. 该别名 Error 可以产生一个错误信息，后面跟着传递给它的所有参数。
3. 变量 file 赋值为从命令行传递给它的第一个参数$1(也就是 testing1)。
4. 如果传递的参数个数为 0，也就是说，如果没有传递任何参数，则别名 Usage 将其信息打印到屏幕。
5. 如果 file(变量替换后)不存在(注意非运算符“!”)，则执行 then 后面的别名 Error 来显示出错信息。
6. 如果 file 是一个目录，则显示“testing1 is a directory”。
7. 如果 file 不是一个目录，而是一个普通文件，那么进入下一层 if 结构进行判断。
8. 该 if 语句嵌套在前一个 if 语句之中。如果 file 可读并且可执行，则执行 then 后面的语句。该 if 语句拥有对应的 endif 并通过对齐的方式来显示其匹配关系。
9. 该 endif 结束了最内层的 if 结构。
10. 该 endif 结束了最外层的 if 结构。

10.6.21 switch 命令

switch 命令是 if-then-else if 结构的一种替代方案。处理多种选择时，使用 switch 命令能让程序更清晰易懂。switch 表达式的值要与关键字 case 后面的标签进行匹配。case 标签可以是常量表达式和通配符。标签以冒号结尾。default 标签是可选的，如果提供了 default 标签，当所有的 case 标签都未能匹配 switch 表达式时，shell 就会执行 default 指定的操作。breaksw 的作用是将执行转到 endsw。如果省略了 breaksw，一旦匹配到某个标签，shell 就会执行这个标签下的所有语句，直至遇到 breaksw 或 endsw。

格式

```
switch  (变量)
case 常量:
 命令
 breaksw
case 常量:
 命令
 breaksw
endsw
```

范例 10-33

(脚本)

```
    #!/bin/csh -f
    # Scriptname: colors
1   echo -n "Which color do you like? "
2   set color = $<
3   switch ("$color")
4   case bl*:
        echo I feel $color
        echo The sky is $color
5       breaksw
```



```
6    case red:           # It is red or is it yellow?
7    case yellow:
8        echo The sun is sometimes $color.
9        breaksw
10   default:
11       echo $color is not one of the categories.
12       breaksw
13   endsw
```

(命令行)

```
% colors
```

(输出)

```
1    Which color do you like? red
8    The sun is sometimes red.
1    Which color do you like? Doesn't matter
11   Doesn't matter is not one of the categories.
```

说明

1. 提示用户输入。

2. 将输入赋值给变量 color。

3. switch 语句计算变量 color 的值。它将对单个词或包含多个词的字符串进行求值，后一种情况中必须要用双引号把这个字符串括起来。

4. 这一行的 case 标签是 bl*，表示要将 switch 表达式与以 bl 开头，后跟任意字符的模式进行匹配。如果用户输入 blue、black、blah、blast 等词，shell 就执行这个 case 标签下的命令。

5. breaksw 将程序控制转到 endsw 语句。

6. 如果 switch 语句与这个标签(red)匹配，程序就开始执行下面的语句，直到在第 9 行遇到 breaksw。第 8 行会被执行，显示“The sun is sometimes red.”。

7. 如果第 6 行不匹配(例如，颜色不是红色)，就测试 yellow 的匹配情况。

8. 当 switch 表达式与 red 和 yellow 这两个标签之一匹配时，执行这一行。

9. breaksw 将程序控制转到 endsw 语句。

10. 如果所有的 case 标签都未能与 switch 表达式匹配上，就会转到 default 标签，用法与 if/elseif/else 结构中的类似。

11. 如果用户输入的内容与之前的所有可能都不匹配，就打印这一行。

12. 这个 breaksw 是可选的，因为 switch 将在此处结束。不过最好还是保留此处的 breaksw，这样，将来要增加更多的分支时，就不会疏漏此处的 breaksw。

13. endsw 结束 switch 语句。

switch 嵌套 switch 语句可以嵌套使用，即 switch 语句和它的分支中可以包含另一个 switch 结构。但必须使用 endsw 以结束 switch 语句。default 标签则不是必需的。

范例 10-34

(脚本)

```
#!/bin/csh -f
```

```
    # Scriptname: systype
    # Program to determine the type of system you are on.
    echo "Your system type is: "
1   set release = (`uname -r`)
2   switch (`uname -s`)
3   case SunOS:
4       switch ("$release")
5       case 4.*:
                echo "SunOS $release"
                breaksw
6       case [5-9].*:
                echo "Solaris $release"
                breaksw
7       endsw
        breaksw
    case HP*:
        echo HP-UX
        breaksw
    case Linux:
        echo Linux
        breaksw
8   endsw
```

（输出）

```
Your system type:
SunOS 4.1.2
```

说明

1. 变量 release 被赋值为 uname -r 的输出，即操作系统版本的发布号。
2. switch 命令计算 uname -s 的输出，即操作系统名。
3. 如果系统类型是 SunOS，则执行第 3 行的 case 命令。
4. 计算变量 release 的值，对各种情况进行匹配。
5. 测试所有发布版本为 4 的 case 命令。
6. 测试所有发布版本为 5~9 的 case 命令。
7. 结束内层的 switch 语句。
8. 结束外层的 switch 语句。

10.6.22 here 文档和菜单

shell 脚本使用 here 文档(参见 9.9.2 节“here 文档”)来创建菜单。它通常与 switch 语句结合起来使用。用户看到菜单后，可以从中选择某个菜单项，然后该选项将与 switch 语句中的 case 标签进行匹配。here 文档减少了所需 echo 语句的数量，从而使得程序可读性更强。

范例 10-35

（脚本）

```
    #! /bin/tcsh
```



```
1   echo "Select from the following menu:"
2   cat << EOF
      1) Red
      2) Green
      3) Blue
      4) Exit
3   EOF
4   set choice = $<
5   switch ("$choice")
6   case 1:
          echo Red is stop.
7         breaksw
    case 2:
          echo Green is go\!
          breaksw
    case 3:
          echo Blue is a feeling...
          breaksw
    case 4:
          exit
          breaksw
    default:
          echo Not a choice\!\!
    endsw
8   echo Good-bye
```

(输出)

```
Select from the following menu:
    1) Red
    2) Green
    3) Blue
    4) Exit
2
Green is go!
Good-bye
```

说明

1. 提示用户从第 2 行 here 文档产生的菜单中进行选择。
2. here 文档用于显示颜色选项列表。
3. EOF 标志着 here 文档的结束。
4. 用户输入被赋给变量 choice。
5. switch 语句对变量 choice 求值，然后将它与 case 中的某个标签匹配。
6. 如果用户选择 1，则显示“Red is stop”，第 7 行的 breaksw 命令将使得程序退出 switch 语句并转到第 8 行。如果用户没有选择 1，则程序从标签 2 开始，以此类推。

10.7 循环命令

使用循环结构可以多次执行相同的语句。C shell 支持两种类型的循环：foreach 循环和 while 循环。如果需要对项目清单(例如文件清单或用户名清单)中所有项目逐一执行某组命令，应使用 foreach 循环。如果要持续执行某一命令，直到满足某个特定条件时终止，则应使用 while 循环。

10.7.1 foreach 循环

foreach 命令后面跟一个变量和一个用圆括号括着的词表。进入第一轮循环时，shell 先将词表中的第一个词赋给变量，并将词表左移一个词，然后进入循环体。shell 执行循环体中的每条命令，直到出现 end 语句为止。之后，控制返回循环顶部。shell 将下一个词赋给变量，然后执行 foreach 行后的命令直到 end，控制再回到 foreach 循环的顶部，再处理下一个词，如此反复。当词表变空时，循环结束。

格式

```
foreach 变量 (词表)
  命令
end
```

范例 10-36

```
1   foreach person (bob sam sue fred)
2       mail $person < letter
3   end
```

说明

1. foreach 命令后面跟了一个变量 person，和用圆括号括着的词表。进入第一轮循环时，变量 person 被赋值为 bob。bob 被赋给 person 后就被(左)移出词表，sam 便成了词表的第一个词。遇到 end 语句后，控制回到循环顶部重新开始，sam 被赋给变量 person。这一过程将一直持续到 fred 被移出词表，这时词表变空，于是循环结束。

2. 第一轮循环将文件 letter 的内容通过电子邮件发给用户 bob。

3. 遇到 end 语句后，循环控制返回到 foreach，词表的下一个元素被赋给变量 person。

范例 10-37

(命令行)

```
% cat maillist
tom
dick
harry
dan
```

(脚本)

```
    #!/bin/csh -f
    # Scriptname: mailtomaillist
1   foreach person (`cat maillist`)
```



```
2       mail $person <<EOF
        Hi $person,
        How are you?  I've missed you. Come on over
        to my place.
        Your pal,
              $LOGNAME@`uname -n`
    EOF
3   end
```

说明

1. shell 在圆括号中执行命令替换。文件 maillist 的内容变成了 foreach 命令的词表。词表(tom、dick、harry、dan)中的名字被依次赋给变量 person。执行完循环语句并遇到 end 后，控制返回 foreach，从词表中移出一个名字，将其赋给变量 person。词表中下一个名字将顶替被移走的名字。这一过程将一直持续到所有名字被移走，词表变空为止。

2. 这里用到了 here 文档。从第一个 EOF 到结尾处的 EOF 之间的输入被送到 mail 程序(注意，最后的 EOF 必须紧挨着左页边，而且前后都不能出现空白符)。这份邮件被发送给名单中的每个人。

3. foreach 循环的 end 语句标志出循环的末尾。控制将返回循环顶部。

范例 10-38

```
1   foreach file (*.c)
2       cc $file -o $file:r
    end
```

说明

1. foreach 命令的词表就是当前目录下名字以.c 结尾的文件清单(即所有 C 源文件)。

2. 清单中的所有文件都将被编译。比如，如果第一个待处理的文件是 program.c，shell 将把 cc 命令行扩展为：

```
cc program.c -o program
```

选项:r 用于将扩展名.c 删除。

范例 10-39

(命令行)

```
1   % runit f1 f2 f3 dir2 dir3
```

(脚本)

```
    #!/bin/csh -f
    # Scriptname: runit
    # It loops through a list of files passed as arguments.

2   foreach arg  ($*)
3     if ( -e $arg ) then
      ...         Program code continues here
      else
      ...         Program code continues here
      endif
4   end
```

```
5    echo "Program continues here"
```

说明

1. 脚本名是 runit。命令行参数是 f1、f2、f3、dir2 和 dir3。

2. $*变量的值为一个从命令行传递来的所有参数(位置参数)列表。foreach 命令将依次处理词表里的各个词：f1、f2、f3、dir2 和 dir3。每次循环，都把表里的第一个词赋给变量 arg。在第一个词赋完值后，把它移去(向左)并把下一个词赋给 arg，直到列表为空为止。

3. 对列表里的各项执行该模块里的命令，直至 end 语句。

4. 词表空后 end 语句终止循环。

5. 循环结束后，程序继续运行。

10.7.2 while 循环

while 循环先对表达式进行计算，若表达式为真(非零)，就执行循环中的命令直到遇到 end 语句为止。然后控制返回到 while 表达式，再次计算该表达式，若仍为真，则再次执行这些命令，以此类推。当 while 表达式为假时，循环结束，控制将转向 end 语句后继续向下执行程序。

范例 10-40

(脚本)

```
     #!/bin/csh -f
1    set num = 0
2    while ($num < 10)
3        echo $num
4        @ num++        # See arithmetic.
5    end
6    echo "Program continues here"
```

说明

1. 变量 num 的初始值设为 0。

2. 进入 while 循环并测试表达式。如果 num 的值小于 10，表达式就是真，则执行第 3 行和第 4 行。

3. 每次进入循环都显示 num 的值。

4. 变量 num 的值递增。如果该语句被省略，循环将永远继续下去。

5. end 语句标志着循环语句的结尾。当到达该行时，控制将返回到 while 循环顶部并再对表达式求值。这将一直持续到 while 表达式为假(即当$num 等于 10 时)。

6. 循环结束后程序从这里继续。

范例 10-41

(脚本)

```
     #!/bin/csh -f
1    echo -n  "Who wrote \"War and Peace\"?"
2    set answer = $<
3    while ("$answer" != "Tolstoy")
         echo "Wrong, try again\!"
```




```
4        set answer = $<
5    end
6    echo Yeah!
```

说明

1. 提示用户输入。
2. 将用户输入的内容赋值给变量 answer。
3. while 命令计算表达式。如果$answer 的值不匹配串“Tolstoy”，则显示消息“Wrong,try again!”，并且再次等待用户输入。
4. 赋给变量 answer 一个新的输入。这一行很重要。如果变量 answer 的值未发生改变，循环表达式就一直为真，从而导致循环无休止地执行下去。
5. end 语句结束本次循环，控制将跳到循环顶部。
6. 如果用户输入“Tolstoy”，循环表达式的值就为假，则控制转到该行，屏幕上显示 Yeah!。

10.7.3 repeat 命令

repeat 命令带两个参数，一个数和一个命令。该命令将被执行该数所指定的次数。

范例 10-42

```
% repeat 3 echo hello
hello
hello
hello
```

说明

执行 echo 命令 3 次。

10.7.4 循环控制命令

shift 命令 如果没有数组名作为它的参数，shift 命令将从左边开始移动 argv 数组中的一个词，因此 argv 数组的长度减 1。一旦移出，该数组元素就丢失了。

范例 10-43

(脚本)

```
    #!/bin/csh -f
    # Scriptname: loop.args
1   while ($#argv)
2       echo $argv
3       shift
4   end
```

(命令行)

```
5   % loop.args a b c d e
    a b c d e
    b c d e
```

```
c d e
d e
e
```

说明

1. $#argv 的值为命令行参数的个数。如果有 5 个命令行参数，a、b、c、d 和 e，则第一次进入循环时$#argv 的值是 5。测试表达式其结果为 5，为真。

2. 打印命令行参数。

3. argv 数组向左移一位。将只剩下 4 个参数，从 b 开始。

4. 到达循环末尾，控制回到循环顶部。重新计算表达式。这次，$#argv 的值是 4。打印参数，并再次移动数组。这样一直持续到所有参数都被移出。此时，当计算表达式时，结果会等于 0，为假，退出循环。

5. 通过 argv 数组把参数 a、b、c、d 和 e 传递给脚本。

break 命令　break 命令可以跳出一个循环，以便控制转向 end 语句后开始执行。如范例 10-44 所示，它跳出最内层的循环。从循环的 end 语句后继续执行。

范例 10-44

```
   #! /bin/csh-f
   # Scriptname: baseball
1  echo -n "What baseball hero died in August 1995? "
2  set answer = $<
3  while ("$answer" !~ [Mm]*)
4           echo "Wrong\! Try again."
            set answer = $<
5           if ("$answer" =~ [Mm*])break
6  end
7  echo "You are a scholar."
```

说明

1. 提示用户输入。

2. 把用户的输入赋给变量 answer(用户输入：Mickey Mantle)。

3. while 表达式可解释为：当 answer 的值不是以一个大写 M 或小写 m 开头，后面跟零个或多个任意字符的字符串时，进入循环。

4. 用户再次输入。变量被重置。

5. 如果变量 answer 与 M 或 m 相匹配，break 就跳出循环。转到 end 语句并开始执行第 7 行的语句。

6. end 语句在循环结束后终止该语句模块。

7. 退出循环后，控制从这里开始并执行该行。

范例 10-45

```
#!/bin/csh -f
# Scriptname: database
```



```
1  while (1)
       echo "Select a menu item"
2      cat << EOF
       1) Append
       2) Delete
       3) Update
       4) Exit
   EOF
3      set choice = $<
4      switch ($choice)
5      case 1:
              echo "Appending"
              break          # Break out of loop; not a breaksw
       case 2:
              echo "Deleting"
              break
       case 3:
              echo "Updating"
              break
       case 4:
              exit 0
6      default:
             echo "Invalid choice. Try again.
       endsw
7  end
8  echo "Program continues here"
```

说明

1. 这种循环称死循环。表达式的值总是 1，即总为真。

2. 这是一个 here 文档。在屏幕上显示一个菜单。

3. 用户选择一个菜单项。

4. switch 命令计算变量。

5. 如果用户输入一个 1~4 之间的数字，则执行相应的 case 标签后的命令。break 语句使程序跳出循环并从第 8 行开始执行。不要把这个跟 breaksw 语句混淆，后者是以 endsw 退出 switch。

6. 如果与 default 情况相匹配，即没有相匹配的 case，将显示 echo 中的信息。之后，程序控制就转向循环末尾，并且再次从循环顶部开始。因为 while 后的表达式总是真，所以将直接进入循环体并再次显示菜单。

7. while 循环语句的结束。

8. 退出循环后，执行该行。

嵌套循环和 repeat 命令 通常为避免使不用 goto 命令，我们采用 repeat 命令来跳出嵌套循环。repeat 命令一般不与 continue 命令结合使用。

范例 10-46

（脚本）

```
    #!/bin/csh -f

1  while (1)
      echo "Hello, in 1st loop"
2     while(1)
            echo "In 2nd loop"
3           while(1)
            echo "In 3rd loop"
4               repeat 3 break
            end
      .end
   end
5  echo "Out of all loops"
```

（输出）

```
Hello, in 1st loop
In 2nd loop
In 3rd loop
Out of all loops
```

说明

1. 开始第 1 轮 while 循环。
2. 进入第 2 层嵌套的 while 循环。
3. 进入第 3 层嵌套的 while 循环。
4. repeat 命令将执行 3 次 break。它将首先跳出最内层循环，然后是第 2 层循环，最后是第 1 层循环。控制从第 5 行继续向下执行。
5. 程序控制在循环终止后从这里开始执行。

continue 命令 continue 语句从最内层循环顶部开始执行。

范例 10-47

```
1   set done = 0
2   while (! $done)
      echo "Are you finished yet?"
      set answer = $<
3     if ("$answer" =~ [Nn]*) continue
4      set done = 1
5   end
```

说明

1. 把 0 赋给变量 done。
2. 测试表达式，它等同于：while(!0)。非 0 的值即代表真(逻辑非)。
3. 如果用户输入 No、no 或 nope(任何以 N 或 n 开头的)，则表达式为真，而且 continue 语句把控制返回到循环顶部，并将重新计算表达式。
4. 如果 answer 不以 N 或 n 开头，则把变量 done 重置为 1。当到达循环尾部时，控制从循环顶部开始并测试表达式。它等同于：while(!1)。非 1 即为假。则退出循环。



5. 标志 while 循环结束。

范例 10-48

(脚本)

```
    #!/bin/csh -f
1   if ( ! -e memo ) then
        echo  "memo file non existent"
        exit 1
    endif
2   foreach person (anish bob don karl jaye)
3      if ("$person" =~ [Kk]arl) continue
4      mail -s "Party time"  $person < memo
    end
```

说明

1. 对文件进行检查。如果文件 memo 不存在，就给用户发一条出错消息，并以状态 1 退出。

2. 循环将依次把表里的每个人都赋给变量 person，然后从表里移去这个名字，继续处理下一个。

3. 如果人名是 Karl 或 karl，continue 语句就从 foreach 循环顶部开始执行(从而不给 Karl 送 memo，因为他的名字在赋给 person 后被移去了)。把表里的下一个名字赋给 person。

4. 给邮件列表里除 karl 外的每个人都发送 memo 文件。

10.8 中断处理

如果用中断键中断了一个脚本，该脚本将终止，于是控制将返回 C shell，重新出现提示符。onintr 命令用来处理一个脚本里的中断。它允许您忽略中断(^C)或在退出前把控制转到程序的其他部分。通常，interrupt 命令带一个标号，用来在退出前进行“清理”(clean up)。不带参数的 onintr 命令则执行默认操作。

范例 10-49

(脚本)

```
1   onintr  finish
2       < Script continues here >
3   finish:
4   onintr -        # Disable further interrupts
5   echo Cleaning temp files
6   rm $$tmp* ; exit 1
```

说明

1. onintr 命令后跟一个标号名。finish 是一个用户自定义的标号。如果发生一个中断，控制将被转到 finish 标号。通常该行位于脚本的开头。除非在脚本里用到它，否则它不会起作用。

2. 除非当程序正在执行时按^C(中断键)，此时，控制被转到该标号，否则将继续执行脚本中其余的行。

3. 这是标号。当中断到来时，程序将继续运行，执行标号下的语句。

4. 要使脚本的这部分屏蔽中断，就使用 onintr 命令。此时如果按下 Ctrl+C 组合键，其作用将会被忽略。

5. 本行回显到屏幕上。

6. 删除所有的 tmp 文件。tmp 文件以 shell 的 PID($$)号为前缀并加上任意个字符的后缀。程序以状态 1 退出。

10.9 setuid 脚本

无论是谁，只要他/她正在运行这个 setuid 程序，此人都将暂时被赋予该程序的所有者权限，即拥有与所有者相同的权限。passwd 程序就是 setuid 程序的一个绝佳的例子。改变口令时，您会暂时成为 root 用户，但是只是在 passwd 程序执行期间。这就是为什么您能在/etc/passwd(或/etc/shadow)文件中更改口令的原因，该文件一般禁止普通用户进行修改。

shell 程序可以被当作 setuid 程序来编写。假定您有一个脚本，它正在访问一个包含这样信息的文件：该信息对普通用户不可访问，如薪水或个人信息。那么你就可能需要编写一个 setuid 脚本。如果脚本是一个 setuid 脚本，则正在运行该脚本的人就可以访问数据，但是仍然会受他人的限制。建立一个 setuid 程序可参照如下步骤：

1. 在脚本中，第一行是：

```
#!/bin/csh -feb
其中，-feb 选项即为
 -f   fast start up; don't execute .cshrc
 -e   abort immediately if interrupted
 -b   this is a setuid script
```

2. 接着，改变脚本权限。这样，它就可以作为一个 setuid 程序来运行。

```
% chmod 4755 script_name
   或
% chmod +srx script_name
% ls -l
-rwsr-xr-x 2 ellie          512 Oct 10 17:18 script_name
```

10.10 保存脚本

成功创建多个脚本后，通常要将它们集中保存到一个脚本目录中，并且修改 path 变量，以便在任何位置都能执行这些脚本。



范例 10-50

```
1   % mkdir ~/bin
2   % mv myscript ~/bin
3   % vi .login

    (在.login 文件中重新设置 path 以加入~/bin)
4       set path = ( /usr/ucb /usr /usr/etc ~/bin . )

5    (命令行)
    % source .login
```

说明

1. 在主目录下创建一个名为 bin 的目录，也可以选用其他名字。

2. 把所有正确的脚本都移到目录 bin 下。若放入错误的脚本只会带来麻烦。

3. 打开.login 文件，修改 path 变量。

4. path 变量的新值中包括目录~/bin，shell 将在 path 变量指定的位置搜索可执行程序。目录~/bin 被放在搜索路径的末尾，因此，如果您的脚本与某个系统程序重名，shell 将执行那个系统程序。

5. 对.login 文件使用 source 命令，这样对 path 所作的改动将生效。而不需要通过注销再重新登录来使之生效。

10.11　内置命令

内置命令不像 UNIX/Linux 命令那样驻留在磁盘上，它们是 C/TC shell 内部代码的一部分，在 shell 内部执行。如果内置命令出现在管道中的任何一个位置(末尾除外)，shell 就会创建一个子 shell 来执行它。请参考表 10-11 所示的内置命令列表，它所列出的命令对 C shell 和 TC shell 均适用。而表 9-25 列出的则是 TC shell 扩展的内置命令。

表 10-11　C/TC shell 内置命令及含义

内 置 命 令	含　义
:	被解释为空命令，不执行任何操作
alias	为命令取一个别名
bg [%job]	当前或指定作业置于后台
break	跳出最内层的 foreach 循环或 while 循环
breaksw	跳出 switch 结构，从 endsw 后继续执行
case label:	switch 语句的标签
cd [dir] chdir [dir]	把 shell 当前目录改为 dir 目录。如果未指定参数，工作目录将改为当前用户的主目录
continue	继续执行最内层的 while 或 foreach 操作
default:	switch 语句中默认情况的标号。default 应当在所有的 case 标号后出现

(续表)

内置命令	含 义
dirs [-l]	显示目录堆栈，最近的在左边。所显示的第一个目录是当前目录。带-l 参数时，就会显示一个详细的打印结果(禁用~符号)
echo [-n] list	把表里的词写入 shell 的标准输出，用空格符分隔。除非使用-n 选项，否则就用一个换行符终止输出
eval command	将命令作为 shell 的标准输入运行，并执行产生的命令。一般执行的是命令或变量替换后所产生的命令，并在这些替换之前进行分析(例如，eval 'test –s option')
exec command	执行命令来取代当前终止的 shell
exit [(expr)]	退出 shell，带状态变量值或由 expr 指定的值
fg [% job]	把当前的或指定的作业置于前台
foreach var (wordlist)	参见 10.7.1 节 foreach 循环
glob wordlist	对单词表执行文件名扩展。像 echo 一样，但是不识别转义符(\)。输出里用空格符来分隔单词
goto label	参见 10.6.14 节 goto 命令
hashstat	显示统计信息以表明内部散列表定位命令(并且避免 exec)的效率有多高。若在路径中的散列函数显示一个可能的命中，就对该路径中的每个组件试用 exec，并在每个不反斜杠开头的组件里试用它
history [-hr] [n]	显示历史清单。如果指定 n，就只显示 n 个最近事件 -r：把打印顺序反转成从最新的开始，而不是从最旧的开始 -h：显示历史清单，没有前导数。用这个来产生适于用-h 选项编写源代码的文件
if (expr)	参见 10.6 节，“条件结构和流控制”
else if (expr2) then	参见 10.6 节，“条件结构和流控制”
jobs [-l]	列出作业控制下的活动作业。-l 列出 ID，外加标准信息
kill [-sig] [pid] [%job] … kill –l	在默认或者有指定信号的情况下，发送 TERM(终止)信号给指定的 ID、指定的作业或当前作业。用编号或者名字来指定信号。信号没有默认值。仅敲入 kill 并不会发送一个信号给当前作业。如果被发送的信号是 TERM(终止)或 HUP(挂起)，那么将也会给作业或进程发送一个 CONT(继续)信号 -l 列出可以被发送的信号名
limit [-h] [resource[max-use]]	限制当前进程和由它创建的每个进程的资源耗费，每个都不能超过指定的最大用量。如果省略了最大用量，就打印当前的极限值。如果省略了资源，就显示所有资源的极限值 -h 则是用硬极限而不是当前极限。硬极限指的是给当前极限值强加一个最高限度。只有超级用户可以提高硬极限。资源分别为：cputime，每个进程所能占用的最多 CPU 秒数；filesize，所允许的文件最大尺寸；datasize，进程的最大数据量(包括栈)；stacksize，进程的最大栈容量；coredump，主存储器信息转储的最大容量；descriptors，文件描述符字数的最大值



(续表)

内置命令	含　义
login [username| -p]	结束登录 shell 并调用 login(1)。不处理.logout 文件。如果省略了用户名，login 就提示输入用户名
	-p 保护当前环境(变量)
logout	终止一个登录 shell
nice [+n | -n][command]	给 shell 或命令的进程优先级值加 n。优先级值越高，进程优先级越低，运行的速度越慢。如果省略了命令，nice 就增加当前 shell 优先级值。如果没有指定递增值，nice 就把值置为 4。nice 值的范围是从-20~19。超出这个范围的 n 则分别置为最低值或最高值
	+n 进程优先级值递增 n
	-n 递减 n。该参数只能由超级用户使用
nohup [command]	忽略 HUP(挂起)命令。没有参数时，就对整个脚本的剩余部分忽略 HUP
notify [%job]	在当前或指定作业的状态改变时异步通知用户
onintr[- | label]	控制 shell 中断操作。若没有参数，onintr 就还原 shell 的中断默认操作(终止 shell 脚本，并返回到终端命令输入层)。有带负号的参数，shell 就忽略所有的中断。若带一个标号参数，shell 就在收到中断或子进程因为中断而终止时执行 goto label 命令
popd[+n]	弹出目录堆栈并返回到新的顶层目录。目录栈的元素从零开始编号，从栈顶开始
	+n 删除栈内第 n 项
pushd[+n | dir]	把一个目录放到目录栈中。若不带参数，就交换顶部的两个元素
	+n 把第 n 项循环到栈顶并 cd 指向它
	dir 当前工作目录入栈并且转为 dir 目录
rehash	重新计算 path 变量中包含目录内容的内部散列表来计入新加的命令
repeat count command	重复执行命令 count 次
set[var [=value]]	参见 9.10 节，“变量”
setenv[VAR[word]]	参见 9.10 节，“变量”。最常用的环境变量，USER、TERM 和 PATH，被自动导入和导出 csh 变量，user、term 和 path；无须对它们使用 setenv。另外，shell 根据 csh 变量 cwd 来设置 PWD 环境变量，而不考虑后者的修改时间
shift[variable]	如果提供了 argv 的数组或变量，就进行左移，删除第一个元素。变量没有置位，或出现空值都会出错
source[-h]name	读取 name 变量指定的命令。source 命令可以嵌套，但是如果嵌套太深，shell 可能会缺少文件描述符。且任何一层的源文件里的任一个错误就会终止所有 source 命令。一般用来重新执行 login 或.cshrc 文件来确保变量的设置都在当前 shell 中处理了，即 shell 不用创建一个子 shell(派生)。-h 用于把从文件名中得到的命令放在历史清单中但不执行它
stop[%job]…	停止当前或指定的后台作业
suspend	在 shell 的栈里停止 shell，就像用^Z 给它发送了一个停止信号。常用于停止 su 启动的 shell

(续表)

内 置 命 令	含 义
switch(string)	参见 10.6.21 节，“switch 命令”
time[Command]	若没有参数，就打印这个 C shell 和它的子进程使用的总时间。带一个可选的命令时，就执行该命令并打印它所用的总时间
umask[value]	显示文件创建掩码。带有值，就设置文件创建掩码。用以八进制形式给出的值跟文件的 666 和目录的 777 权限相异或，来得到新文件的权限。而不能通过 umask 把权限相加
unalias pattern	删除与模式相匹配(文件名替换)的别名。而 unalias.*则删除所有的别名
unhash	删除内部散列表
unlimit[-h][resource]	删除一个资源限制。如果没有指定资源，就删除所有的资源限制。参见 limit 命令中关于资源名列表的描述
	-h 删除相应的硬极限。只有超级用户能这样做
unset pattern	删除名字与模式相匹配(文件名替换)的变量。而 unset *则删除所有变量，这很有可能产生不良后果
unsetenv variable	从环境中删除变量。就跟用 unset 一样，但不执行模式匹配
wait	在提示前等待后台作业完成(或等待一个中断)
while(expr)	参见 10.7.2 节 “while 循环命令”

习题 24 C/TC shell 入门

1. init 进程可以执行哪些操作？
2. login 进程的功能是什么？
3. 如何判断正在用什么 shell？
4. 如何改变登录 shell？
5. 解释.cshrc/.tcshrc 和.login 文件之间的不同之处。先执行哪一个？
6. 编辑.chsrc/.tcshrc 文件如下。
   a) 创建 3 个自己的别名。
   b) 重置提示符。
7. 设置下列变量并在每个变量后添加一个注释来解释它所做的工作。

```
noclobber # Protects clobbering files from redirection overwriting
history
ignoreeof
savehist
filec
```

8. 键入 source .cshrc、source .tcshrcsource 命令后输出什么？
9. 编辑.login 文件如下：
   a) 欢迎用户。
   b) 如果工作目录不在路径中就把它加上。
   c) 对.login 文件执行 source 命令。



10. 键入 history，输出什么？

a) 怎样重新执行最后一条命令？

b) 现在键入：echo a b c，并用 history 命令来重新执行只带最后一参数 c 的 echo 命令。

习题 25 shell 元字符

1. 在提示下敲入：

```
touch ab abc a1 a2 a3 a11 a12 ba ba.1 ba.2 filex filey AbC ABC ABc2 abc
```

2. 写出相应命令并进行测试。

a) 列出所有以 a 开头的文件。

b) 列出所有末尾至少有一个数字的文件。

c) 列出所有以一个 a 或 A 开头的文件。

d) 列出所有以一个句点后跟一位数字结尾的文件。

e) 列出所有只包含两个字母 a 的文件。

f) 列出所有 3 个字符都大写的文件。

g) 列出末尾是 11 或 12 的文件。

h) 列出以 x 或 y 结尾的文件。

i) 列出所有以一个数字，一个大写字母，或一个小写字母结尾的文件。

j) 列出所有包含一个 b 的文件。

k) 删除以 a 开头的两个字符的文件。

习题 26 重定向

1. 与终端相关联的 3 个文件流的名称是什么？
2. 什么是文件描述符？
3. 写出相应命令：

a) 把 ls 命令的输出重定向到文件 lsfile。

b) 把 date 命令的输出重定向并添加到 lsfile。

c) 把 who 命令的输出重定向到 lsfile。会产生什么结果？

4. 只敲入 cp 时会出现什么？

a) 怎么把上面例子中的出错消息保存到一个文件里？

5. 用 find 命令来查找所有始于父目录，且是“目录”类型的文件。把标准输出保存在一个名为 found 的文件里并把任何错误都保存在文件 found.errs 里。
6. 什么是 noclobber？怎么忽略它？
7. 取得 3 个命令的输出并把输出重定向到文件 gotemail 里。
8. 把管道和 ps 及 wc 命令一起使用来找出当前正在运行的进程个数。

习题 27 第一个脚本

1. 编写一个名为 greetme 的脚本，让它执行下列操作：

a) 问候用户。

b) 打印当前日期和时间。

c) 打印这个月的日历。
d) 打印机器名。
e) 打印父目录中所有文件的列表。
f) 打印正在运行的所有进程。
g) 打印变量 TERM、PATH 和 HOME 的值。
h) 打印“Please could you loan me $50.00?”
i) 跟用户说“Good bye”并且告诉他当前时间(请参考 date 命令的手册页)。

2. 确保您的脚本可执行。

```
chomd +x greetme
```

3. 您的脚本中第一行内容是什么？

习题 28 读取用户输入

1. 写一个名为 nosy 的脚本，该脚本将执行下列操作：
a) 询问用户的全名——名和姓。
b) 用用户的名问候他(她)。
c) 询问用户的出生年份，并计算出他(她)的年龄。
d) 询问用户的登录名，并打印他(她)的用户 ID (从/etc/passwd 中获取)。
e) 告诉用户他(她)的主目录所在位置。
f) 向用户显示他(她)正在运行的进程。
g) 告诉用户现在是星期几，并且用 12 小时制的时间格式告诉他(她)现在的时间。输出结果应类似于：

```
The day of the week is Tuesday and the current time is 04:07:38 PM.
```

2. 创建一个名为 datafile 的文本文件。文件中每条记录包含由冒号分隔的若干字段。记录包含如下字段：
a) 名和姓
b) 电话号码
c) 地址
d) 出生日期
e) 工资

3. 创建一个名为 lookup 的脚本，要求完成如下任务：
a) 包含一节注释，介绍脚本名、作者姓名、时间和编写这个脚本的原因。编写这个脚本的原因是要排序显示 datafile 的内容。
b) 按姓氏对 datafile 排序。
c) 向用户显示 datafile 的内容。
d) 告诉用户文件中一共有多少条记录。

4. 尝试用 echo 和 verbose 命令来调试脚本。如何使用这些命令？

习题 29 命令行参数

1. 写一个名为 rename 的脚本，让它执行下列操作：
a) 用两个文件名作为命令行参数，第一个是文件的原名，第二个则是新文件名。



b) 用一个新文件名重命名旧文件。

c) 列出当前目录中的文件以显示前一操作带来的变化。

2. 写一个名为 checking 的脚本，让它执行下列操作：

a) 取得一个命令行参数，一个用户的注册名。

b) 测试一下看看是否提供了一个命令行参数。

c) 查看用户是否在/etc/passwd 文件里。如果在，将打印：

```
"Found <user> in the /etc/passwd file."
```

否则，就打印：

```
"No such user on our system."
```

习题 30 条件语句与文件测试

1. 在 lookup 脚本中，询问用户是否愿意给 datefile 增加一项记录。如果回答 yes 或 y，则完成以下操作：

a) 提示用户输入一组新的名字、电话、地址、生日和工资。每项都分别存在一个单独的变量中。并且要在各个字段间加冒号并把信息添加到 datefile 中。

b) 按姓氏给文件排序。通知用户新添加了一条记录，并在前面加上行号以显示该行。

2. 改写习题 29 中的 checking 脚本。要求检查指定的用户是否在/etc/passwd 文件中，接着检查这个用户是否已登入系统。如果是，程序就打印出正在运行的所有进程。否则，程序将告诉用户：

```
<用户名> is not logged on.
```

3. 脚本 lookup 的运行需要基于 datafile 文件。在脚本 lookup 中，检查文件 datafile 是否存在，是否可读并且可写。

4. 在脚本 lookup 中增加一个菜单：

```
[1] Add entry
[2] Delete entry
[3] View entry
[4] Exit
```

5. 已完成的脚本中将存在 Add entry 这部分。现在要往 Add entry 中添加代码，检查用户提供的姓名是否已在文件 datafile 中出现，如果是，就通知用户；否则，就增加这条新记录。

6. 现在为删除记录(Delete entry)、查看记录(View entry)和退出(Exit)函数编写代码。

7. 脚本中处理删除的部分应该首先检查记录是否存在，然后才去删除它。如果记录不存在，则通知用户这个错误。否则，就删除这条记录，并且告诉用户记录已被删除。退出时，一定要返回一个数字来代表相应的退出状态。

8. 试着从命令行检查退出状态。

习题 31 switch 语句

1. 用一条 switch 语句重写下列脚本。

```
# !/bin/csh -f
# Grades program
echo -n "what was your grade on the test? "
set score=$<
if ($grade >= 90 && $grade <= 100) then
```

```
      echo you got an A\!
   else if ($grade >= 80 && $grade <= 89) then
      echo you got a B
 else if ($grade >= 70 && $grade <= 79) then
     echo "you're average. "
 else if ($grade >= 60 && $grade <=69) then
     echo Better study harder
 else
     echo Better luck next time.
 endif
```

2. 对每个菜单项使用 switch 语句来重写 lookup 脚本。

习题 32 循环

1. 写一个名为 picnic 的程序，它将依次给一组用户邮寄一封野餐的邀请函，一次发给一个人。用户列表保存在一个名为 friends 的文件里。在 friends 文件中有一个名为 Popeye 的用户。

a) 邀请函保存在另一个名为 invite 文件里。

b) 用文件测试来检查文件是否存在以及可读。

c) 将用一个循环来对用户列表进行遍历。当遇到 Popeye 时，就跳过他(即，他收不到邀请函)，并给表里的下一个用户发一封邀请函，以此类推。

d) 保存一个收到邀请函的人名列表。建立一个数组来实现这一功能。在给表里的每个人都发完邮件后，打印收到邮件的人数和名单。

附：如果时间允许，还可以定制 invite 文件以便每个用户都能收到一封显示自己姓名的信。例如消息可以这样开头：

```
Dear John,
I hope you can make it to our picnic....
```

要在 invite 文件中做到这一点可以这样写：

```
Dear XXX,
I hope you can make it to our picnic....
```

然后，用 sed 或 awk，再用用户名替换 XXX 即可(把大写字母放在用户名位置是一种技巧，因为用户名总是小写的)。

2. 添加一个如下所示的新菜单到 lookup 脚本中。

```
[1]Add entry
[2]Delete entry
[3]Change entry
[4]View entry
[5]Exit
```

当用户选择了一个有效项后，函数完成时将询问用户是否愿意再次查看菜单。如果输入了一个无效项，程序就打印：

```
Invalid entry,try again.
```



然后重新显示菜单。

3. 在 lookup 脚本里，在 View entry 下面创建一个子菜单。用于询问用户是否愿意查看一个已选中人的详细信息。

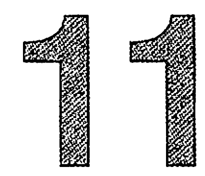
a) 电话

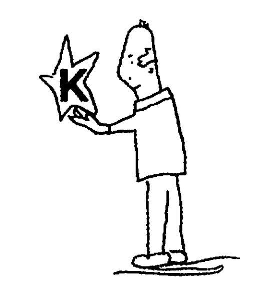
b) 地址

c) 生日

d) 工资

4. 用一个标号把 onintr 命令加到脚本中。当程序从标号处开始执行时，将删除所有临时文件，将对用户显示 Good-bye，然后退出程序。





chapter 11 交互式 Korn shell

11.1 简介

在交互式 shell 中，标准输入、输出和错误输出都是绑定到终端上的。因此，在交互式使用 Korn(ksh)时，用户需在 ksh 提示符下键入 UNIX/Linux 命令，并等待终端响应。交互式 Korn shell 结合了 UNIX Bourne 和 C shell 的优点，提供了内置命令与命令行快捷键。例如历史命令、别名、文件和命令补全等。David Korn 还对 shell 进行了扩展，包含了更多的功能，如 vi 和 emacs 命令行编辑、新的元字符、协同处理及错误处理等。尽管 Korn shell 发源于 AT&T，但是应用广泛，可以在很多操作系统上运行。

本章将集中讨论如何在命令行上与 ksh 进行交互，以及如何定制工作环境。您将学会通过利用快捷键和内置特征，来创建高效和用户友好的运行环境。第 12 章将进一步讨论有关 ksh 的编程问题。学习完第 12 章之后，您将可以编写出 ksh shell 脚本，并通过裁剪自己的工作环境，更好地自动完成每天的任务，开发出复杂的脚本。如果您是系统管理员，那么这些工作的完成将不仅方便自己，也能方便所有其他用户。

启动

在 Korn shell 显示提示符之前，会首先执行几个进程(参见图 11-1)。系统要运行的第一个进程是 init，它的 PID 是 1。它从文件 inittab 中读取指令(System V 的做法)，或者派生一个 getty 进程(BSD 的做法)。这些进程打开终端端口，以提供标准输入(stdin)的来源及标准输出(stdout)的去处，还提供标准错误(stderr)输出的去处，并且在屏幕上显示一个登录提示符。接下来执行的是/bin/login 程序，提示用户输入口令、加密并验证用户输入的口令、设置初始环境、启动用户的登录 shell。用户的登录 shell 是 passwd 文件中每一条记录的最后一项，本章是/bin/ksh。ksh 程序运行后，它首先查找系统文件/etc/profile，并且执行其中的命令。然后在用户的主目录下查找初始化文件.profile 和环境文件.kshrc，并执行其中的命令。执行完以上文件中的命令后，屏幕上显示提示符，即美元符号($)，然后 ksh 开始等待用户输入命令。



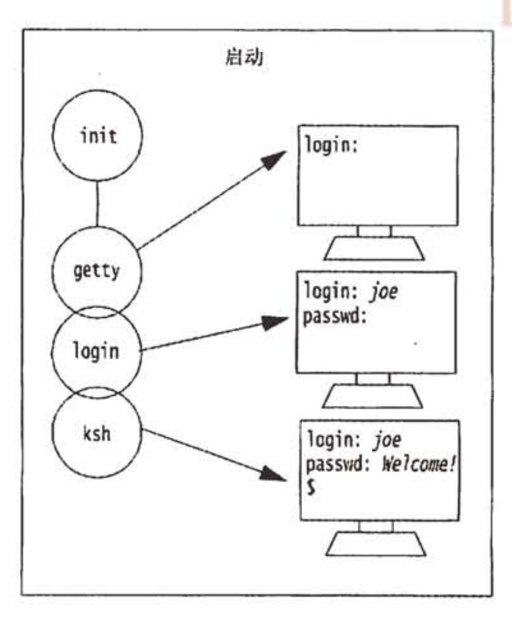


图 11-1 启动 Korn shell

11.2 环境

11.2.1 初始化文件

执行完系统文件/etc/profile 中的命令后，ksh 接着执行用户主目录下的初始化文件：先执行.profile 文件，然后是 ENV 文件，通常命名为.kshrc 文件。

etc/profile 文件 /etc/profile 文件是一个系统可读的文件，由系统管理员进行设置，使用户在登录和 ksh 启动时执行指定的任务。它可以被系统上的所有 Bourne shell 和 Korn shell 用户使用，通常执行诸如邮件查收、显示文件/etc/motd 中的当日信息之类的任务。下面是一个/etc/profile 文件的例子，可参见第 8 章中的相关章节，其中对该文件中的每一行都有完整的解释。

范例 11-1

```
# The profile that all logins get before using their own .profile
   trap " " 2 3
   export LOGNAME PATH  # Initially set by /bin/login
   if [ "$TERM" = " " ]
   then
      if /bin/i386
      then    # Set the terminal type
         TERM=AT386
      else
         TERM=sun
      fi
```

```
    export TERM
fi
# Login and -su shells get /etc/profile services.
# -rsh is given its environment in the .profile.
case "$0" in
-sh | -ksh | -jsh )
   if [ ! -f .hushlogin ]
   then
      /usr/sbin/quota
   # Allow the user to break the Message-Of-The-Day only.
       trap "trap ' ' 2" 2
       /bin/cat -s /etc/motd
       # Display the message of the day
       trap " " 2
       /bin/mail -E
       case $? in
       0)   # Check for new mail
           echo "You have new mail. "
           ;;
      2)    echo "You have mail. "
           ;;
      esac
   fi
esac
umask 022
trap 2 3
```

.profile 文件 .profile 是用户定义的初始化文件，在主目录下，它会在用户登录时执行一次(由 Bourne shell 和 Korn shell 执行)。这个文件提供了定制和修改 shell 环境的功能。环境和终端的设置通常都放在这个文件中，此外，如果要初始化某个窗口应用程序或数据库应用程序，也是在这里进行。如果文件中包含一个特殊的变量 ENV，则赋给它的值是一个文件名，该文件接下来将被执行。ENV 文件通常命名为.kshrc，它包含 alias 和 set –o 命令，每派生一个 ksh 子 shell 就执行一次。下面是一个 ENV 文件的例子，也许您目前还不能弄明白每一行的意思，但我们将随着本章内容的展开，详细讨论所有的概念，如导出变量、命令历史、搜索路径等。

范例 11-2

```
1   set -o allexport
2   TERM=vt102
3   HOSTNAME=$(uname -n)
4   HISTSIZE=50
5   EDITOR=/usr/ucb/vi
6   ENV=$HOME/.kshrc
7   PATH=$HOME/bin:/usr/ucb:/usr/bin:\
    /usr/local:/etc:/bin:/usr/bin:/usr/local\
    /bin:/usr/hosts:/usr/5bin:/usr/etc:/usr/bin:.
8   PS1="$HOSTNAME ! $ "
9   set +o allexport
10  alias openwin=/usr/openwin/bin/openwin
```



```
11  trap '$HOME/.logout' EXIT
12  clear
```

说明

1. 通过设置 allexport 选项，所有创建的变量都将自动导出(对所有 subshell 可见)。

2. 终端设置为 vt102。

3. 变量 HOSTNAME 赋值为机器名$(uname -n)。

4. 变量 HISTSIZE 赋值为 50。即用户键入 history 命令时，历史文件将显示 50 行的内容。

5. 变量 EDITOR 赋值为 vi 编辑器的路径。mail 等程序可以选择工作所需的编辑器。

6. 变量 ENV 赋值为 home 目录($HOME)的路径以及包含 Korn shell 定制设置的文件名。.profile 文件执行后，将执行 ENV 文件。ENV 的文件名可任选，通常用作为.kshrc 文件名。

7. 定义搜索路径。shell 会用以分号隔开的目录在提示符下搜索命令，或者在脚本运行时搜索路径。shell 在路径中从左到右搜索命令。最后的句点表示当前工作目录。如果在所列出的目录中找不到命令，shell 将查看当前目录。

8. 主提示符(默认为美元符号$)被设置为主机名，历史文件中当前命令的编号以及一个美元符号($)。

9. 关闭 allexport 选项。

10. 别名是命令的简称。openwin 的别名被赋为 openwin 命令的完整路径名，该命令将启动 Sun 的窗口应用程序。

11. 退出 shell 时，trap 命令将执行.logout 文件。.logout 文件是一个用户定义的文件，它包含在注销时要执行的命令。例如，用户可能希望记录注销的时间、删除临时文件、或者只是简单地显示“So long”。

12. clear 命令用来清屏。

ENV 文件　ENV 变量所指定的文件，在每次启动交互式的 ksh 或 ksh 程序(脚本)时执行。ENV 变量在.profile 文件中指定。ENV 文件中包含特殊的 ksh 变量和别名，其文件名通常为.kshrc，也可以是其他的文件名(当特权(privileged)选项为打开状态时，ENV 文件将不执行。参见表 11-1)。

范例 11-3

```
1  $ export ENV=.kshrc
2  $ cat .kshrc
   # The .kshrc file
3  set -o trackall
4  set -o vi
5  alias l='ls -laF'
   alias ls='ls -aF'
   alias hi='fc -l'
   alias c=clear
6  function pushd { pwd > $HOME/.lastdir.$$ ; }
   function popd { cd $(< $HOME/.lastdir.$$) ;
            rm $HOME/.lastdir.$$; pwd; }
```

```
function psg {.ps -ef | egrep $1 | egrep -v egrep; }
function vg { vgrind -s11 -t $* | lpr -t ; }
```

说明

1. ENV 变量赋值为每次调用 Korn shell 时所执行的文件的文件名。导出 ENV 将使得它对所有的子 shell 可见。
2. 显示了.kshrc 文件的内容。
3. 打开用来跟踪别名的 set 选项(完整的描述参见 11.5 节的“别名”)。
4. 打开用于 vi 编辑器的 set 选项，以进行历史文件的行内编辑(参见 11.3.6 节的“命令行历史”)。
5. 定义了命令的别名。
6. 对函数进行命名和定义(参见 11.12 节的“函数”)。

set –o 选项　set 命令用来设置位置参数(参见 11.9 节的“变量”)。使用 - o 选项时，set 命令可以设置 ksh 的选项。用户可以使用选项来定制 shell 环境，选项要么开(on)，要么关(off)，通常在 ENV 文件中进行设置(参见表 11-1)。

格式

```
set -o option     Turns on the option.
set +o option     Turns off the option
set -[a-z]        Abbreviation for an option; the minus turns it on
set +[a-z]        Abbreviation for an option; the plus turns it off
```

范例 11-4

```
1 set -o allexport
2 set +o allexport
3 set -a
4 set +a
```

说明

1. 设置 allexport 选项，所有变量被自动导出到子 shell 中。
2. 复位 allexport 选项设置，所有变量都变成当前 shell 的局部变量。
3. 设置 allexport 选项，作用与第 1 行一样。但并不是每一个选项都有缩写(参见表 11-1)。
4. 复位 allexport 选项设置，作用与第 2 行一样。

表 11-1　Korn shell 的 set 选项

选项名称	缩写	作用
allexport	- a	设置变量自动导出
bgnice		后台任务在更低优先级下运行
emacs		使用 emacs 内置编辑器来编辑命令行中的命令
errexit	- e	设置时，当命令返回非零值(失败)时，执行 ERR 陷阱(若设置)后再退出。读初始化文件时，不进行设置
gmacs		使用 gmacs 内置编辑器编辑命令行命令



(续表)

选项名称	缩写	作用
ignoreeof		禁止用^D(Ctrl+D组合键)登出，必须键入exit才能退出shell
markdirs		当使用文件名扩展时，在目录名后跟一个反斜杠(/)
monitor	-m	允许作业控制
noclobber		当使用重定向时，确保文件不被改写
noexec	-n	读入命令但不执行，以检查脚本语法。在非交互式运行时，不设置该选项
noglob	-f	禁止路径名扩展，即关闭通配符
nolog		在历史文件中不保存函数定义
notify		在后台任务结束时通知用户
nounset		在扩展未定义的变量时显示一条错误信息
privileged	-p	一旦设置，shell将不读取.profile或ENV文件。在setuid脚本中使用
trackall		使别名跟踪生效
verbose	-v	调试时打开verbose模式
vi		使用vi内置编辑器来编辑命令行中的命令
xtrace	-x	调试时打开echo模式

11.2.2 提示符

Korn shell 提供了4种提示符，主提示符和次提示符，它们将在Korn shell交互式运行时使用。用户可以改变提示符。变量PS1代表主提示符，最初设为美元符号($)。变量PS2代表次提示符，最初设为字符“>”，它在用户输入命令不全、按下回车键时出现。用户可以改变主提示符和次提示符的设置。

主提示符 美元符号“$”是默认的主提示符。用户可以改变提示符。通常是在.profile文件中定义提示符。

范例 11-5

```
1   $ PS1="$(uname -n) ! $ "
2   jody 1141 $
```

说明

1. 默认的主提示符是“$”。PS1提示符被重置为机器名$(uname -n)、在命令预留文件中当前命令的序号以及一个美元符号。感叹号表示当前命令预留的个数(在PS1定义中用两个感叹号(!!)可打印出一个感叹号)。

2. 显示出新的提示符。

次提示符 PS2提示符是次提示符，它的值将显示到标准错误输出上(即屏幕)。当用户输入的命令尚未完成并按下回车时，将显示该提示符。

范例 11-6

```
1  $ print "Hello
2  > there"
3  Hello
   there
4  $
5  $ PS2="----> "
6  $ print "Hi
7  ------>
   ------>
   ------> there"
   Hi

   there
   $
```

说明

1. 双引号必须成对出现，字符串“Hello 的后面缺少配对的双引号。
2. 回车后到新的一行，次提示符出现了，并一直出现，直到输入用于闭合的双引号。
3. 显示 print 命令的输出。
4. 显示主提示符。
5. 重置次提示符。
6. 双引号必须成对出现，字符串“Hi 的后面缺少配对的双引号。
7. 到新的一行后，次提示符出现了，并一直出现，直到输入用于闭合的双引号。

11.2.3 搜索路径

在命令行或 shell 脚本中执行所输入命令时，Korn shell 将在 PATH 变量所列的目录中搜索该命令。PATH 是一个用冒号分隔的目录列表，shell 在所列的目录中从左到右依次搜索，最后的句点表示当前工作目录。如果在搜索路径的目录列表中找不到要找的命令，shell 就向标准错误输出一条信息：ksh: filename not found。建议在.profile 文件中设置搜索路径。为加速搜索过程，Korn shell 实现了别名跟踪，参见 11.5.4 节的相关内容。

范例 11-7

```
$ echo $PATH
/home/gsa12/bin:/usr/ucb:/usr/bin:/usr/local/bin:.
```

说明

Korn shell 从/home/gsa12/bin 目录开始搜索命令，如果在此目录找不到命令，接着在/usr/ucb 目录中搜索，然后依次是/usr/bin，/usr/local/bin，最后是在用户当前工作目录中搜索。

11.2.4 句点命令

句点命令(.)是 Korn shell 的一个内置命令，该命令将一个脚本文件名作为命令参数，该脚本在当前 shell 环境下执行时不启动子进程。句点命令通常在.profile 或 ENV 文件修改



后，重新执行它们。例如，如果登录后以上文件中的设置改变了，用户可以用句点命令重新执行这两个初始化文件，而不必登出系统后再登录进来。

范例 11-8

```
$ . .profile
$ . .kshrc
$ . $ENV
```

说明

通常，执行一个命令时，会启动一个子进程。句点命令是在当前 shell 中执行初始化文件.profile 或 ENV 文件(.kshrc)，文件中定义的局部和全局变量都是当前 shell 的变量。否则，用户只能注销退出系统后再登录进入系统来执行这些文件，句点命令则省去了这些麻烦。

11.3 命令行

用户登录成功后，Korn shell 会显示它的主提示符，默认情况下是一个美元符号。现在，Korn shell 就是命令解释器。以交互方式运行时，shell 从终端读取命令，把命令行分解为若干个词。一个命令行由一个或多个词(标记)组成，词与词之间用空白符(空格或制表符)分隔。命令行以换行符结束，按一次回车键产生一个换行符。命令行的第一个词是命令，后续的词是命令的参数。命令可以是一个 UNIX 可执行程序(比如 ls 和 pwd)，也可以是 shell 的一条内置命令(比如 cd 和 jobs)，或某个 shell 脚本。命令行中可能含有称作元字符的特殊字符，shell 分析命令行时必须翻译这些元字符。如果命令行很长，用户可以输入一个反斜杠，接着换行，就可以转到新行继续输入命令，这时新行将显示次提示符，直到命令行结束之前，shell 会在接下来的每一行上显示次提示符。

11.3.1 命令执行的次序

命令行的第一个词是要执行的命令，命令可以是一个关键字、特殊的内置命令或工具、函数、脚本，或是一个可执行程序，各种命令根据其类型按以下顺序执行[①]：

(1) 关键字(如 if，while，until)

(2) 别名(见 typeset －f)

(3) 内置命令

(4) 函数

(5) 脚本和可执行程序

特殊的内置命令和函数是在 shell 内部定义的，运行时从当前 shell 中开始执行，因此它们的执行速度更快。脚本和可执行程序(如 ls，pwd)存储在磁盘上，shell 首先必须通过 PATH 环境变量在层式目录中定位它们，然后生成一个新 shell 来运行它。要知道所用命令的类型，可以用内置命令 whence –v，或是它的别名 type(参见 11-9)。

① 内置命令优先于函数，因此，必须为函数定义别名(参见 11.5 节的“别名”)。在 Korn shell 的 1994 版本中，处理函数与内置命令的顺序是相反的，因此部分地解决了这个问题。

范例 11-9

```
$ type print
print is a shell builtin
$ type test
test is a shell builtin
$ type ls
ls is a tracked alias for /usr/bin/ls
$ type type
type is an exported alias for whence -v
$ type bc
bc is /usr/bin/bc
$ type if
if is a keyword
```

11.3.2 退出状态

命令或程序终止后，会向父进程返回一个退出状态。退出状态是一个 0~255 之间的整数。按照惯例，程序退出时，如果返回的状态是 0，表示命令执行成功；如果退出状态非 0，则表示命令因某种原因而执行失败。Korn shell 的状态变量(?)被设置为 shell 执行的最近一条命令的退出状态值。程序运行结果是成功还是失败，由编写它的程序员来确认。在 shell 脚本中，可以用 exit 命令显式地控制退出状态。

范例 11-10

```
1  $ grep "ellie" /etc/passwd
   ellie:GgMyBsSJavd16s:9496:40:Ellie Quigley:/home/jody/ellie
2  $ echo $?
   0

3  $ grep "nicky" /etc/passwd
4  $ echo $?
   1

5  $ grep "scott" /etc/passsswd
   grep: /etc/passsswd: No such file or directory
6  $ echo $?
   2
```

说明

1. grep 程序在/etc/passwd 文件中搜索模式 ellie，若程序执行成功，则输出/etc/passwd 中的对应行。
2. 变量“?”被置为 grep 命令的退出状态。值为 0 表示成功。
3. grep 命令在文件/etc/passwd 中没有找到用户 nicky。
4. grep 程序如果找不到模式，就会返回退出状态 1。
5. grep 因为打不开文件/etc/passsswd 而运行失败。
6. grep 如果找不到文件，就会返回退出状态 2。



11.3.3 含多条命令的命令行和命令组

一个命令行可以包括多条命令。命令之间用分号隔开，命令行以换行符终止。

范例 11-11

```
$ ls; pwd; date
```

说明

从左到右逐一地执行命令，直至遇到换行符。可以把多条命令聚成一组，以便把所有命令的输出通过管道发给另一个命令或者重定向到文件。

范例 11-12

```
$ (ls ; pwd ; date ) >outputfile
```

说明

每个命令的输出都发送到文件 outputfile。

11.3.4 命令的条件执行

有条件地执行命令时，要用特殊的元字符，即双与号(&&)或双竖杠(||)来间隔两个命令串。是否执行元字符右侧的命令，取决于左侧命令的退出状态。

范例 11-13

```
$ cc prgm1.c -o prgm1 && prgm1
```

说明

如果第一条命令执行成功(退出状态为 0)，就执行&&后面的命令。

范例 11-14

```
$ cc prog.c >& err || mail bob < err
```

说明

即使第一条命令执行失败(退出状态不为 0)，|| 后面的命令也要执行。

11.3.5 后台执行的命令

命令通常都在前台运行，要等到命令执行完之后，提示符才会重新出现。有时候要等待命令结束会不太方便，通过在命令行的末尾加上一个与号(&)，shell 就会立即返回提示符，同时在后台执行这条命令。这样，不需要等待就能执行另一条命令。后台任务会在执行过程中将它产生的输出随时发送到屏幕上。因此，如果您想在后台运行一条命令，不妨将它的输出重定向到某个文件，或者通过管道发给某个设备(比如打印机)，这样，后台命令的输出结果就不会干扰您正在前台进行的工作。

范例 11-15

```
1   $ man ksh | lp&
2   [1] 1557
3   $
```

说明

1. 通过管道将 Korn shell 的帮助信息发送给打印机。命令行末尾的与号把作业放入后台执行。

2. 屏幕上输出两个数字：方括号里的数字表示这是第一个被放入后台的作业，第二个数是一个 PID，是该作业的进程标识号。

3. Korn shell 提示符立刻就出现了。程序在后台运行的同时，shell 开始等待下一条在前台运行的命令。

11.3.6 命令行历史

历史机制把在命令行键入的命令记录在一个历史文件中。可以从历史文件中再次调入前面键入的命令，并执行命令，而不需要再次键入该命令。Korn shell 的内置命令 history 可以显示出历史列表。命令历史文件默认的文件名为.sh_history，存放在用户的主目录下。

变量 HISTSIZE 指定了从命令历史文件可以访问的命令数，ksh 第一次访问命令历史文件时将访问该变量，其默认值是 128。HISTFILE 变量指定了命令历史文件的文件名(默认文件名是~/.sh_history)，命令都将存放在该文件中。历史文件在一次登录会话期间，逐渐增长，直到用户清理之前，文件会变得非常大。命令 history 是命令 fc –l 的预设别名。

范例 11-16

```
(~/sh_history 文件)
(此文件包含了用户在命令行输入的多条命令)

    netscape&
    ls
    mkdir javascript
    cd javascript
    &cp ../javapdf.zip .
    gunzip javapdf.zip
    unzip javapdf.zip
    ls
    more chapter10*
    ls -l
    rm chapter9.pdf
    ls
    ls -l

    ... continues ...

1   $ history -1 -5         # List last 5 commands, preceding this one in
                            # reversed order.
    13  history -3
    12  history 8
    11  history -n
    10  history
    9   set
```



```
2   $ history -5 -1        # Print last 5 commands, preceding this one in order.
    10   history
    11   history -n
    12   history 8
    13   history -3
    14   history -1 -5
3   $ history              # (Different history list)
    78   date
    79   ls
    80   who
    81   echo hi
    82   history
4   $ history ls echo      # Display from most recent ls command to
    79   ls                # most recent echo command.
    80   who
    81   echo hi
5   $ history -r ls echo   # -r reverses the list
    81   echo hi
    80   who
    79   ls
```

history 命令/重新显示命令 内置的 history 命令按数字顺序列出前面键入的命令，它也可以通过参数来设定所显示的内容和格式。

范例 11-17

```
1   $ history              # Same as fc -l
    1 ls
    2 vi file1\
    3 df
    4 ps -eaf
    5 history
    6 more /etc/passwd
    7 cd
    8 echo $USER
    9 set
    10 history
2   $ history -n           # Print without line numbers
    ls
    vi file1
    df
    ps -eaf
    history
    more /etc/passwd
    cd
    echo $USER
    set
    history
    history -n
3   $ history 8            # List from 8th command to present
    8    echo $USER
    9    set
```

```
    10   history
    11   history -n
    12   history 8
4   $ history -3         # List this command and the 3 preceding it
    10   history
    11   history -n
    12   history 8
    13   history -3
5   $ history -1 -5     # List last 5 commands, preceding this one in reversed
                        # order.
    13   history -3
    12   history 8
    11   history -n
    10   history
    9    set
6   $ history -5 -1     # Print last 5 commands, preceding this one in order.
    10   history
    11   history -n
    12   history 8
    13   history -3
    14   history -1 -5
7   $ history           # (Different history list)
    78   date
    79   ls
    80   who
    81   echo hi
    82   history
```

用 r 命令重复执行命令　r 命令重复执行在命令行键入的最后一条命令。如果 r 命令后紧跟一个空格和数字，就再次执行该数字所在行的命令。如果 r 命令后紧跟一个空格和字母，就执行以该字母开头的最后一条命令。不带任何参数时，r 命令就重复执行历史列表中的最近一次的命令。

范例 11-18

```
1   $ r date
    date
    Mon May 15 12:27:35 PST 2004
2   $ r 3    redo command number 3
    df
    Filesystem   kbytes    used      avail     capacity   Mounted on
    /dev/sd0a    7735      6282      680       90%        /
    /dev/sd0g    144613    131183    0         101%       /usr
    /dev/sd2c    388998    211395    138704    60%        /home.
3   $ r vi          # Redo the last command that began with pattern vi.
4   $ r vi file1=file2 # Redo last command that began with vi and substitute
                       # the first occurrence of file1 with file2.
```

说明

1. 重新执行最近执行的命令 date。
2. 执行历史文件中的第 3 条命令。



3. 重新执行最近一条以字符串 vi 开头的命令。
4. 字符串 file1 被 file2 替换，最近的命令 vi file1 也被替换成 vi file2。

11.3.7 命令行编辑

Korn shell 提供两个内置的编辑器：emacs 和 vi，用户可以以交互方式编辑历史列表。要设置使用 vi 编辑器，使用 set 命令。把下面给出的 set 命令行添加到.profile 文件中即可。内置的 emacs 编辑器每次只能编辑历史文件中的一行，而内置的 vi 编辑器可以同时编辑多行。若设置使用 vi 编辑器，则键入：

```
set -o vi 或 VISUAL=vi 或 EDITOR=/usr/bin/vi
```

如果使用 emacs 编辑器，则键入：

```
set -o emacs 或 VISUAL=emacs 或 EDITOR=/usr/bin/emacs
```

注意，set –o vi 优先于 VISUAL，而 VISUAL 优先于 EDITOR。

内置的 vi 编辑器 要编辑历史列表，按下 Esc 键后就可以使用 vi 的标准键来进行上下移动、左右移动、删除、插入和修改文本。参见表 11-2。完成编辑后，按下 Enter 键，命令就被执行并添加到历史列表的底部。要在历史文件中向上翻卷，按 Esc 键后再按 K 键。

表 11-2 vi 命 令

命 令	功 能
在历史文件中移动	
Esc k 或+	在历史列表中上移一行
Esc j 或 -	在历史列表中下移一行
Esc G	移动到历史文件的第 1 行
Esc 5G	移动到历史文件的第 5 行命令
/string	在历史文件中从前向后搜索字符串
?	在历史文件中从后向前搜索字符串
在一行内移动(首先按下 Esc 键)	
h	在一行中向左移动
l	在一行中向右移动
b	回退一个单词
e 或 w	前进一个单词
^或 0	移动到行首：一行的第一个字符
$	移动到行尾
用 vi 编辑	
a A	追加文本
i I	插入文本
dd dw x	删除文本(一行、一个单词或一个字符)，放到缓冲区内
cc C	改变文本

(续表)

命　令	功　能
u　U	撤销
yy Y	复制一行到缓冲区
p　P	将复制或删除到缓冲区的行放到当前行的前面或后面
r　R	替换一行上的一个或多个字符

内置的 emacs 编辑器　在历史文件中向上移动，按^P 键(即 Ctrl+P 组合键)。向下移动，按^N 键(Ctrl+N 组合键)。使用 emacs 编辑器改变文本或纠正文本错误，然后按 Enter 键，命令就被重新执行。见表 11-3。

表 11-3　emacs　命　令

命　令	功　能
在历史文件中移动	
Ctrl + P	在历史列表中上移一行
Ctrl + N	在历史列表中下移一行
Ctrl + B	回退一个字符
Ctrl + R	从后向前搜索字符串
Esc B	回退一个单词
Ctrl + F	向前移动一个字符
Esc F	向前移动一个单词
Ctrl + A	移动到行首
Ctrl + E	移动到行尾
Esc <	移动到历史文件的第一行
Esc >	移动到历史文件的最后一行
用 emacs 编辑	
Ctrl + U	删除一行
Ctrl + Y	把删除的行取回来
Ctrl + K	从光标位置删除到行尾
Ctrl + D	删除一个字母
Esc D	向前删除一个单词
Esc H	向后删除一个单词
Esc space	在光标所在位置设置一个标记
Ctrl + X Ctrl + X	交换光标和标记
Ctrl + P Ctrl + Y	把光标到标记所在的区域送到缓冲区中(Ctrl + P)，并复制该区(Ctrl + Y)

FCEDIT 和编辑命令　fc 命令是一个内置命令，与 FCEDIT 变量[2] (通常在.profile 文件

② 在比 1988 版本新的 Korn shell 中，FCEDIT 变量被重命名为 HISTEDIT，而 fc 命令被重命名为 hist。



中设置)一起使用，来调用用户所选择的用来编辑历史文件的编辑器。编辑器可以是系统中的任何编辑器，FCEDIT 变量设置为编辑器的完整路径名，如果未设置 FCEDIT，那么键入 fc 命令时就调入默认编辑器/bin/ed。

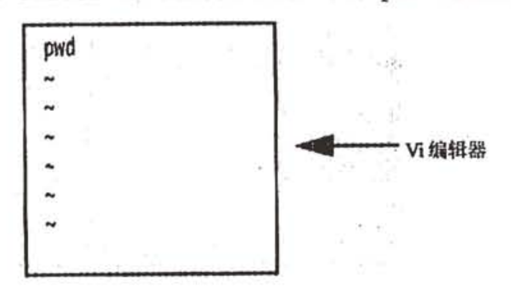
FCEDIT 可以被设置为所选的编辑器，用户也可以指定想要编辑的历史列表中的命令数，完成编辑后，Korn shell 将执行整个文件。以#号开头的命令，都被视为注释，不被执行。要了解更多有关注释和文件名扩展的信息，请参见表 11-4。

表 11-4 使用 Esc 键和文件名扩展

组 合	结 果
command [Esc]#	“#”：用符号#来引导命令，命令将不执行，只是作为注释放到历史列表中
command [Esc]_	下划线：将最后一条命令的最后一个单词插入到当前光标位置
command [Esc]2_	将最后一条命令的第二个单词插入到当前光标位置
word[Esc]*	“*”：用匹配的所有文件替换当前单词
word[Esc]\	“\”：用第一个以某个字符开始的文件名替换当前单词，用于文件名扩展
word[Esc]=	“=”：显示出所有以当前单词开头的文件名，显示格式为一个按数字排序的列表

范例 11-19

```
1   $ FCEDIT=/usr/bin/vi
2   $ pwd
3   $ fc
<Starts up the full-screen vi editor with the pwd command on line 1>
```




```
4   $ history
    1 date
    2 ls -l
    3 echo "hello"
    4 pwd

5   $ fc -3 -1    # Start vi, edit, write/quit, and execute last 3 commands.
```

说明

1. 变量 FCEDIT 可以被设置为系统上的任何 UNIX 文本编辑器，如 vi、emacs、textedit 等，默认的编辑器是 ed。

2. 从命令行键入了命令 pwd，它将保存在历史文件中。

3. 命令 fc 调用编辑器(编辑器由变量 FCEDIT 设置)，最后一个命令显示在编辑器窗口中。当用户在编辑器中写下命令后，在退出编辑器时，其中的所有命令都将被执行。

4. 命令 history 列出了最近键入的命令。
5. 命令 fc 启动编辑器，并把历史文件中的最近 3 条命令调入到编辑器的缓冲区内。

11.4 文件名扩展

文件名扩展是一种有用的机制，它允许用户只键入文件名的一部分，再按下 Esc 键后，就可以看到文件名的剩余部分。在下面的例子中，[Esc]代表 Esc 键。

范例 11-20

([Esc]即按下 Esc 键)

```
1   $ ls a[Esc]=
    1) abc
    2) abc1
    3) abc122
    4) abc123
    5) abc2
2   $ ls a[Esc]*
    ls abc abc1 abc122 abc123 abc2
    abc   abc1  abc122 abc123 abc2
3   $ print apples pears peaches
    apples pears peaches
4   $ print [Esc]_
    print peaches
    peaches
5   $ print apples pears peaches plums
    apples pears peaches
6   $ print [Esc]2_
    print pears
    pears
```

说明

1. 键入字符 a，再按 Esc 键，再键入等号(=)，所有以字母 a 开头的文件名都将显示为一个按数字排序的列表(其中的数字没有任何特殊作用)。
2. 键入字符 a，再按 Esc 键，再键入星号(*)，将显示所有以字母 a 开头的文件名。
3. 命令 print 显示它的 3 个参数：apples、pears、peaches。
4. Esc 键后紧跟一个下划线(_)，该组合将被最后一个参数替换。
5. 命令 print 显示它的 3 个参数：apples、pears、peaches。
6. Esc 键后紧跟数字 2 和一个下划线(_)，该组合将被第二个参数替换。命令(print)本身是第一个参数，参数索引从 0 开始。

11.5 别名

别名是 Korn shell 或用户定义的一个命令的缩写。别名可以包括字母和数字。一个命



令的默认别名由 shell 定义，也可以由用户重新定义或取消定义。与 C Shell 的别名不同，Korn shell 不支持别名传送参数(如果需要用到参数，参见 11.12 的“函数”一节)。

把别名保存在 ENV 文件中，就可以在子 shell 中使用它们(每派生一个新 shell，ENV 文件中的命令都要执行一次)。在 1998 版的 Korn shell 中，只要新 shell 不是一个单独调用的 ksh，用–x 选项就可以使得别名在该新 shell 中可用。Korn shell 还支持别名定位，以提高 shell 搜索路径的速度。别名自己还可以有别名，也就是说，别名是可以递归的。

11.5.1 别名列表

内置命令 alias 将列出所有已设置的别名。

范例 11-21

```
1  $ alias
2  autoload=typeset -fu
3  false=let 0
4  functions=typeset -f
5  hash=alias -t
6  history=fc -l
7  integer=typeset -i
8  r=fc -e -
9  stop=kill -STOP
10 suspend=kill -STOP $$
11 true=:
12 type=whence -v
```

说明

1. 不带参数的 alias 命令，列出所有的别名，即一个预设别名的列表，包括所有已经设置的别名。

2. 别名 autoload：用来动态地调用函数。

3. 别名 false：用在表达式测试中，表示为否定条件。

4. 别名 functions：列出所有的函数及其定义。

5. 别名 hash：列出所有的别名定位。

6. 别名 history：按顺序列出命令历史文件.sh_history 中的所有命令并编号。

7. 别名 integer：用来创建一个整型的变量。

8. 别名 r：使用户重新执行历史列表中的前一条命令。

9. 别名 stop：如果 kill 命令带一个任务号或 PID，则挂起该进程。在前台使用命令 fg，可使挂起的任务恢复运行。

10. 别名 suspend：向 kill 命令发送 STOP 信号和当前进程的 PID($$)，从而挂起当前任务。

11. 别名 true：设为空操作命令，通常用来执行一个无限循环。

12. 别名 type：显示一条命令的类型：包括别名类型、二进制可执行类型等。

11.5.2 创建别名

用户可以在 Korn shell 中创建别名。别名通常是现有命令或多条命令的简称，当 shell 分析命令行时，将用真正的命令替换别名。

范例 11-22

```
1   $ alias cl='clear'
2   $ alias l='ls -laF'
3   $ alias ls='ls -aF'
4   $ \ls ..
```

说明

1. 别名 cl：是命令 clear 的别名。
2. 别名 l：是命令“ls - laF”的别名。
3. 别名 ls：设置为命令“ls - aF”。
4. 反斜杠表示不用别名来执行本命令行，而是执行 ls 命令本身。

11.5.3 删除别名

用 unalias 命令可以删除别名。

范例 11-23

```
unalias cl
```

说明

将别名 cl 从别名列表中删除。

11.5.4 别名定位

为了减少在路径中搜索所需的时间，Korn shell 在第一次碰到某个命令时将创建一个别名，该别名等价于一个完整路径的命令，这就是别名定位[③]。

Korn shell 有一些预设的在安装时就定义好的别名定位。要使用别名定位，执行命令 set –o trackall，该命令通常设在 ENV 文件中。要查看所有的别名定位，执行命令 alias - t。

范例 11-24

```
$ alias -t
chmod=/bin/chmod
ls=/bin/ls
vi=/usr/ucb/vi
who=/bin/who
```

说明

内置命令 alias 带上 - t 选项，将列出所有通过定位机制设置的别名。当用户键入这些命令时，shell 直接使用别名定义来调用命令，而不用从路径中搜索命令。

---

③ 如果重置 PATH 变量，将不会定义别名定位。



11.6 作业控制

作业控制用来控制前台作业和后台作业的执行。

要使用 Korn shell 的作业控制，在不支持作业控制的系统上必须设置监控选项(set –o monitor)。作业控制命令见表 11-5。

表 11-5 作业控制命令

命 令	功 能
jobs	按作业号的顺序列出所有未完成的进程，其中作业号列在括号中
jobs –l	同 jobs 命令，但是包含作业的 PID
^Z	停止当前作业
fg %n	在前台运行后台作业
bg %n	在后台运行作业
wait %n	等待作业号为 n 的作业结束
kill %n	终止作业号为 n 的作业

范例 11-25

```
1   $ vi
    [1]  + Stopped          # vi
2   $ sleep 25&
    [2] 4538
3   $ jobs
    [2] + Running           # sleep 25&
    [1] - Stopped           # vi
4   $ jobs -l
    [2] + 4538   Running    # sleep 25&
    [1] - 4537   Stopped    # vi
5   $ fg %1
```

说明

1. 当调用 vi 编辑器后，可以按下^Z(即 Ctrl+Z 组合键)来挂起 vi 会话，此时编辑器将在后台挂起。出现 Stopped 消息之后，将显示 shell 提示符。

2. 命令尾的&符号表示将在后台执行 sleep 命令，参数为 25。标记[2]表示这是在后台运行的第二条命令，其 PID 为 4538。

3. 命令 jobs 显示当前在后台运行的作业。

4. 带 - l 选项的 jobs 命令显示当前在后台运行的作业及其 PID。

5. 命令 fg 带一个百分号和一个作业号，表示把该作业从后台提到前台。不带数字参数时，fg 将把最近一个后台的作业提到前台。

11.7 元字符

元字符是代表自身以外的内容的特殊字符。表 11-6 列出了一些普通的 Korn shell 元字符及其功能。

表 11-6 Korn shell 的元字符

元 字 符	功 能
\	按字面含义解释它后面那个字符
&	在后台处理
;	分隔命令
$	替换变量
?	匹配单个字符
[abc]	匹配这一组字符中的一个
[!abc]	匹配这一组字符以外的某个字符
*	匹配零个或多个任意字符
(cmds)	在子 shell 中执行命令
{cmds}	在当前 shell 中执行命令

范例 11-26

```
1   $ ls -d *    all files are displayed
    abc   abc122   abc2   file1.bak file2.bak nonsense  nothing one
    abc1 abc123   file1   file2    none    noone

2   $ print hello \      # The carriage return is escaped
    > there
    hello there

3   $ rusers&           # Process the rusers command in the background
    [1]   4334
    $

4   $ who; date; uptime    # Commands are executed one at a time
    ellie  console Feb 10 10:46
    ellie  ttyp0    Feb 15 12:41
    ellie  ttyp1    Feb 10 10:47
    ellie  ttyp2    Feb 5  10:53
    Mon Feb 15 17:16:43 PST 2004
    5:16pm up 5 days, 6:32, 1 user, load average: 0.28, 0.23, 0.01

5   $ print $HOME         # The value of the HOME variable is printed
   /home/jody/ellie

6   $ print $LOGNAME      # The value of the LOGNAME variable is printed
    ellie
```




```
7   $ ( pwd; cd / ; pwd )
    /home/jody/ellie
    /
    $ pwd
    /home/jody/ellie

8   $ { pwd; cd /; pwd; }
    /home/jody/ellie
    /
    $ pwd
    /

9   $ ( date; pwd; ls ) > outfile
    $ cat outfile
    Mon Feb 15 15:56:34 PDT 2004
    /home/jody/ellie
    foo1
    foo2
    foo3
```

说明

1. 星号匹配当前目录下的所有文件。ls 命令的 -d 选项禁止显示子目录的内容。
2. 使用反斜杠可以在下一行继续输入以完成一条完整的命令。
3. &符号使 rusers 程序在后台执行，并立即返回 shell 提示符。这样，两个进程就可以同时运行。
4. 每个命令都由分号隔开，每次只执行一个命令。
5. 如果一个美元符号后紧跟一个变量，则 shell 将执行变量替换。显示的是环境变量 HOME 的值，即用户主目录的完整路径。
6. 同上，美元符号后紧跟一个变量。变量 LOGNAME 的值是用户的登录名。
7. 圆括号表示括在其中的命令将在子 shell 中执行。cd 命令是 shell 的内置命令，因此每个调用的 shell 都有自己的 cd 命令副本。pwd 命令显示的是当前工作目录 /home/jody/ellie。在运行命令 cd 切换目录到根目录后，pwd 命令显示的是新的工作目录，即根目录。在子 shell 退出后，父 shell 的 pwd 命令输出的是/home/jody/ellie，仍然是进入子 shell 前的目录。
8. 花括号表示包含在其中的目录将在当前 shell 中执行。pwd 命令显示出当前工作目录/home/jody/ellie。切换到根目录后，pwd 命令显示新的工作目录是根目录。当花括号中的命令退出后，pwd 命令输出的仍然是根目录。
9. 括号用来把命令绑在一起，使所有 3 条命令的输出都被发送到 outfile 文件中。

11.8 文件名替换(通配符)

分析命令行时，shell 会用元字符来表示能够匹配某个特定字符组的文件名或路径名，元字符也称为通配符。表 11-7 中所列的文件名替换元字符将被展开为一组按字母顺序索引

的文件名。将元字符展开为文件名的过程称为文件名替换或 globbing。如果没有文件名能够跟所用的元字符匹配，Korn shell 就会把这个元字符作为一个字面字符。

表 11-7 Korn shell 元字符和文件名替换

元 字 符	含 义
*	匹配零个或多个字符
?	匹配一个字符
[abc]	匹配 a、b、c 这组字符中的一个
[!abc]	匹配除 a、b、c 以外的任一字符
[a - z]	匹配 a 至 z 范围内的某个字符
~	~ 替换成用户主目录
\	转义或禁用后面那个元字符

11.8.1 星号

星号是一个通配符，它匹配文件名中零个或多个任意字符。

范例 11-27

```
1   $ ls *
    abc abc1 abc122 abc123 abc2 file1 file1.bak file2 file2.bak none
    nonsense noone nothing nowhere one
2   $ ls *.bak
    file1.bak file2.bak
3   $ print a*c
    abc
```

说明

1. 星号代表的是当前工作目录下的所有文件名。因此 ls 命令显示的是所有文件的名称。
2. 匹配并列出所有以零个或多个字符开头、以.bak 结尾的文件名。
3. 匹配所有以 a 开头、后跟零个或多个字符、以 c 结尾的的文件名，并将它们作为参数传给 print 命令。

11.8.2 问号

问号代表文件名中某一单个字符。当文件名中包含一或多个问号时，shell 把问号替换为文件名中所匹配的字符。

范例 11-28

```
1   $ ls
    abc abc1 abc122 abc123 abc2 file1 file1.bak file2 file2.bak
    none nonsense noone nothing nowhere one
2   $ ls a?c?
    abc1 abc2
```



```
3   $ ls ??
    ?? not found
4   $ print abc???
    abc122 abc123
5   $ print ??
    ??
```

说明

1. 列出当前目录下的文件。

2. 文件名包含 4 个字符：以 a 开头，后跟一个字符，再跟字符 c 和一个字符。匹配并列出这些文件名。

3. 文件名正好由两个字符组成时，列出这些文件名。因为当前目录下没有名称为两个字符的文件，所以这两个问号被解释为由两个字面字符“?”组成的文件名，而当前目录下没有名为“??”的文件，最后 shell 输出信息“?? not found”。

4. 文件名包含 6 个字符：以 abc 开头、后跟 3 个字符，匹配并列出这些文件名。

5. ksh 的 print 函数把两个问号作为参数，shell 先寻找正好匹配为两个字符的文件名，没有找到，于是 shell 就把问号当成一个字面字符，并作为参数传送给 print 命令。

11.8.3 方括号

方括号用于匹配包含指定字符组或字符范围内任一字符的文件名。

范例 11-29

```
1   $ ls
    abc abc1 abc122 abc123 abc2 file1 file1.bak file2 file2.bak
    none nonsense noone nothing nowhere one
2   $ ls abc[123]
    abc1 abc2
3   $ ls abc[1-3]
    abc1 abc2
4   $ ls [a-z][a-z][a-z]
    abc one
5   $ ls [!f-z]???
    abc1 abc2
6   $ ls abc12[2-3]
    abc122 abc123
```

说明

1. 列出当前工作目录下的所有文件。

2. 跟所有包含 4 个字符的文件名比较，列出以 abc 开头，后跟 1、2 或 3 的文件名，只能选方括号中的一个字符与文件名进行匹配。

3. 跟所有包含 4 个字符的文件名比较，列出以 abc 开头，后跟一个 1~3 之间的任一数字的文件名。

4. 跟所有包含 3 个字符的文件名比较，列出由 3 个小写字母组成的文件名。

5. 列出所有包含 4 个字符、第 1 个字符不是 f 至 z 之间的小写字母，后面是 3 个任意

字符(???)的文件名。

6. 列出文件名以 abc12 开头，后跟 2 或 3 的文件。

11.8.4 转义元字符

如果想把元字符当作字面字符使用，可以用反斜杠来禁止 shell 对它进行解释。

范例 11-30

```
1   $ ls
    abc file1 youx
2   $ print How are you?
    How are youx
3   $ print How are you\?
    How are you?
4   $ print When does this line \
    > ever end\?
    When does this line ever end?
```

说明

1. 列出当前工作目录下的所有文件，注意 youx 这个文件。

2. shell 对“?”执行文件名扩展。匹配当前目录下所有以 you 开头，后面有且只有一个字符的文件名，把它们替换到字符串中。文件名 youx 将被替换到字符串中，字符串将变成：How are youx (这可能不是您想要的结果)。

3. 在问号前面加一个反斜杠，问号就被转义了，这意味着 shell 不会再把问号当作一个通配符，也就不再解释它。

4. 在换行符前面加一个反斜杠将其转义。次提示符将一直出现直到字符串以换行符结尾。问号被转义了，即不对它进行文件名替换。

11.8.5 代字符号和连字符扩展

Korn shell 采用了 C shell 的作法，也把代字符号作为一个路径名扩展。代字符号本身代表用户主目录的完整路径，当代字符号后跟一个用户名时，就被扩展为该用户主目录的完整路径。

连字符号代表此前的工作目录，OLDPWD 所指的也是此前的工作目录。

范例 11-31

```
1   $ echo ~
    /home/jody/ellie
2   $ echo ~joe
    /home/joe
3   $ echo ~+
    /home/jody/ellie/perl
4   $ echo ~-
    /home/jody/ellie/prac
5   $ echo $OLDPWD
    /home/jody/ellie/prac
```



```
6  $ cd -
   /home/jody/ellie/prac
```

说明

1. 代字符号代表用户主目录的完整路径。
2. 代字符号后跟一个用户名，代表用户 joe 主目录的完整路径。
3. 标记 ~+ 代表当前工作目录的完整路径。
4. 标记 ~ - 代表此前工作目录的完整路径。
5. 变量 OLDPWD 包含此前工作目录的完整路径。
6. 连字符号指的是此前工作目录，cd 命令用来切换到此前的工作目录，并显示此目录。

11.8.6 新增的 ksh 元字符

新的 Korn shell 元字符用于文件名扩展，类似于 egrep 和 awk 的正则表达式的元字符。括号中字符前的元字符，控制着模式匹配，参见表 11-8。

表 11-8 正则表达式通配符

正则表达式	含 义
abc?(2|9)1	"?"将匹配括号中的零个或一个模式，竖线表示或，即 2 或 9。匹配后的结果是 abc21、abc91 或 abc1
abc*([0 - 9])	"*"将匹配括号中的零个或多个模式，匹配结果是 abc 后跟零个或多个数字，如 abc1234、abc3、abc2 等
abc+([0 - 9])	"+"将匹配括号中的一个或多个模式，匹配结果是 abc 后跟一个或多个数字，如 abc3、abc123 等
no@(one|ne)	"@"将匹配括号中的一个模式，匹配结果是 noone 或 none
no!(thing|where)	"!"将匹配除了括号中所列模式以外的所有字符串，匹配结果是 no、nobody 或 noone，但不会是 nothing 或 nowhere

范例 11-32

```
1  $ ls
   abc abc1 abc122 abc123 abc2 file1 file1.bak file2 file2.bak none
   nonsense noone nothing nowhere one
2  $ ls abc?(1|2)
   abc  abc1 abc2
3  $ ls abc*([1-5])
   abc   abc1  abc122 abc123 abc2
4  $ ls abc+([1-5])
   abc1  abc122 abc123 abc2
5  $ ls no@(thing|ne)
   none   nothing
6  $ ls no!(one|nsense)
   none  nothing nowhere
```

说明

1. 列出当前工作目录下的所有文件。

2. 匹配的文件名是以 abc 开始，后面跟零个字符，或跟括号中的任一个模式。匹配结果是 abc、abc1 或 abc2。

3. 匹配的文件名是以 abc 开始，后面跟零个数字，或跟 1~5 之间的多个数字。匹配后的结果列表是 abc1、abc122、abc123 和 abc2。

4. 匹配的文件名是以 abc 开始，后面跟一个或多个 1~5 之间的数字，匹配后的结果列表是 abc1、abc122、abc123 和 abc2。

5. 匹配的文件名以 no 开始，后面跟 thing 或 ne，匹配结果是 nothing 或 none。

6. 匹配的文件名以 no 开始，后面跟除了 one 和 nsense 以外的任何字符串，匹配后的结果列表是 none、nothing 和 nowhere。

11.8.7 noglob 变量

如果设置了 noglob 变量，那么将关闭文件名替换，这意味着所有的元字符将代表其自身，而不作为通配符使用。在使用 grep、sed 或者 awk 这些程序来搜索包含元字符的模式时，这种机制非常有用。如果没有设置 noglob 变量，要想元字符不被解释，那就必须在所有的元字符前加上反斜杠。

范例 11-33

```
1  % set -o noglob        # or set -f
2  % print * ?? [] ~ $LOGNAME
   * ?? [] /home/jody/ellie ellie
3  % set +o noglob        # or set +f
```

说明

1. 设置 noglob 变量，关闭所有对文件名扩展有特殊意义的通配符。也可以使用 –f 选项使命令产生同样的结果。

2. 文件名扩展元字符显示为它们本身，不经过任何解释。注意，~ 与 $ 仍然被扩展。

3. 重置 noglob 选项，扩展文件名元数据。

11.9 变量

11.9.1 局部变量

赋给局部变量的值，仅仅对于创建它们的 shell 程序可见。变量名必须以字母或者下划线开头。其余的字符可以是字母、十进制数字 0~9 或者下划线。所有的其他字符，标志着变量名的结束。

设置并引用局部变量

当给一个局部变量赋值时，等号两边不能出现空格。设置一个变量为空时，等号右边



什么都不带即可。如果设置一个局部变量为多个单词，那么必须将这些单词包含在引号中，否则 shell 将显示出错信息，提示这些变量没有定义。

在变量前面加一个美元符号，就可以提取变量中的值。如果需要在变量值后附加其他的字符，那么仅仅需要将变量名用大括号括起来就可以了。

范例 11-34

```
1   $ state=Cal
    $ echo $state
    Cal
2   $ name="Peter Piper"
    $ echo $name
    Peter Piper
3   $ x=
    $ echo $x
          # Blank line appears when a variable is either unset or set to null
    $
4   $ state=Cal
    $ print ${state}ifornia
    California
```

说明

1. 将变量 state 赋值为 Cal。如果遇到变量名前面是美元符号的情况，shell 就执行变量替换。该命令显示变量 state 的值。

2. 变量 name 被赋值为"Peter Piper"。必须用引号来隐藏空白符，这样，shell 分析命令行时才不会将这个字符串分解为两个词。该命令显示变量 name 的值。

3. 这条命令没有给变量 x 赋值，因此 x 被赋为空值。结果显示一个空值，即空字符串。

4. 变量 state 被赋值为 Cal，变量用大括号括起来，以使其与其他的字符屏蔽开。这样变量的值与之后附加的 ifornia 将被显示。

局部变量的作用域　局部变量只能被创建它的 shell 识别。shell 不会将局部变量传给子 shell。而双美元符号是一个特殊变量，它的值为当前 shell 的 PID。

范例 11-35

```
1   $ echo $$
    1313
2   $ round=world
    $ echo $round
    world
3   $ ksh      # Start a subshell
4   $ echo $$
    1326
5   $ echo $round

6   $ exit       # Exits this shell, returns to parent shell
7   $ echo $$
    1313
```

```
8   $ echo $round
    world
```

说明

1. 双美元符号变量的值是当前 shell 的 PID。本例中这个 shell 的 PID 是 1313。
2. 将局部变量 round 赋值为字符串 world，并输出该变量的值。
3. 另外启动一个 Korn shell，这个 shell 被称为子 shell(subshell 或 child shell)。
4. 当前这个 shell 的 PID 是 1326，其父 shell 的 PID 则是 1313。
5. 变量 round 在这个 shell 中没有定义，因此输出了一个空行。
6. exit 命令终止当前进程，返回父进程(若未设置 ignoreeof 选项，按下 Ctrl+D 组合键也能退出当前进程)。
7. 返回到父进程，显示它的 PID。
8. 显示变量 round 的值。

设置只读变量　只读变量不能被重新定义或复位。而只能被内置命令 readonly 或者 typeset –r 设置。通常，当在保护模式下运行 shell 程序时，出于安全因素需要将变量设置为只读。

范例 11-36

```
1   $ readonly name=Tom
    $ print $name
    Tom
2   $ unset name
    ksh: name: is read only
3   $ name=Joe
    ksh name: is read only
4   $ typeset -r PATH
    $ PATH=${PATH}:/usr/local/bin
    ksh: PATH: is read only
```

说明

1. 局部变量 name 的值被设为 Tom。
2. 将变量 name 设为只读。
3. 不能重新定义只读变量。
4. PATH 变量被设置为只读的，因此试图复位或修改这个变量的值都将产生错误消息。

11.9.2 环境变量

环境变量对创建它们的 shell 和该 shell 派生的所有子 shell 和进程都可见。按照惯例，环境变量应该大写。

变量被创建时所处的 shell 称为父 shell。如果父 shell 又启动了一个 shell，这个新的 shell 称为子 shell。有一些环境变量，比如 HOME、LOGNAME、PATH 和 SHELL，在用户登录之前就已经被/bin/login 程序设置好了。通常情况下，环境变量被定义和保存在用户主目录下的.profile 文件中。



设置环境变量 为了设置环境变量，需要在变量设置时或者赋值后，用 export 命令导出该变量。使用 set 命令设置 allexport 选项后，一个脚本中的所有变量都将被导出。

范例 11-37

```
1   $ TERM=wyse ; export TERM
2   $ export NAME ="John Smith"
    $ print $NAME
    John Smith
3   $ print $$
    319
4   $ ksh
5   $ print $$
    340
6   $ print $NAME
    John Smith
7   $ NAME="April Jenner"
    $ print $NAME
    April Jenner
8   $ exit
9   $ print $$
    319
10  $ print $NAME
    John Smith
```

说明

1. 将变量 TERM 设置为 wyse，然后将其导出。现在，由这个 shell 启动的所有进程都将继承这个变量。
2. 同时定义并导出变量 NAME(Korn shell 中的新特性)。
3. 显示当前 shell 的 PID 的值。
4. 启动一个新的 Korn shell。这个新 shell 被称为子 shell，原来那个 shell 被称为父 shell。
5. 新的 Korn shell 的 PID 保存在变量$$中。显示这个变量的值。
6. 在父 shell 中设置的变量 NAME 被导出到这个新 shell 中，并显示它的值。
7. 将变量 NAME 重置为“April Jenner”并显示。这个变化将被导出到所有的子 shell，但不会影响父 shell。
8. 退出这个 Korn 子 shell。
9. 再次显示父 shell 的 PID，即 319。
10. 变量 NANE 的值仍是初始值。从父 shell 导出到子 shell 时，变量保持它们的值不变。子 shell 不可能改变父 shell 的变量的值。

特殊的环境变量 Korn shell 将环境变量 PATH、PS1、PS2、PS3、PS4、MAILCHECK、FCEDIT、TMOUT 以及 IFS 设置为默认的值。而环境变量 SHELL、LOGNAME、USER 以及 HOME 则在/bin/login 程序中设置。可以修改默认的值，或者设置表 11-9 中列出的环境变量。

表 11-9 Korn shell 环境变量

变 量 名	作 用
_(下划线)	前一条命令的最后一个参数
CDPATH	cd 命令的搜索路径。若路径名不以/、./、../开头，则使用冒号分隔的目录列表来搜索路径
COLUMNS	在指定的命令、编辑模式的 shell 中设置编辑窗口的宽度
EDITOR	内置编辑器的路径名：emacs、gmacs 或者 vi
ENV	设置的一个文件名，该文件包含在 ksh 脚本运行时需要调用的函数以及别名。在 1988 年之后的版本中，只有在交互模式下调用 ksh 该文件才发挥作用，非交互时不起作用。如果特权选项被打开，那么这个变量将不被扩展
ERRNO	系统错误号。它的值为最近一次失败的系统调用编号
FCEDIT	fc 命令默认的编辑器名。在 1988 年之后的版本中，这个变量被称为 HISTEDIT，fc 命令也变成了 hist 命令
FPATH	一个用冒号分隔的目录名列表，它们定义了包含自动加载函数的目录的搜索路径
HISTEDIT	在 1988 年之后的版本中，FCEDIT 的新名称
HISTFILE	存储历史记录的特殊文件
HISTSIZE	最多可保存的历史命令数目，默认为 128
HOME	主目录。未指定参数的 cd 命令将以它为目标
IFS	内部字段分隔符，通常是空格、制表符和回车。用于将命令替换、循环结构中的列表产生的词组分隔成字段，或在读取输入时使用
LINENO	脚本中的当前行号
LINES	在 select 循环中，垂直显示的菜单项数，默认为 24
MAIL	如果该参数被设置为某个邮件文件的名称，而 MAILPATH 未被设置，当邮件到达 MAIL 指定的文件时，shell 会通知用户
MAILCHECK	该参数定义 shell 将隔多长时间(以秒为单位)检查一次由参数 MAILPATH 或 MAIL 指定的文件，看看是否有邮件到达。默认值是 600 秒(10 分钟)。如果将它设为 0，shell 每次输出主提示符之前都会检查邮件
MAILPATH	由冒号分隔的文件名列表。如果设置了这个参数，只要有邮件到达任何一个由它指定的文件，shell 都会通知用户。每个文件名后面都可以跟一个百分号和一条消息，当修改时间发生变化时，shell 会显示这条消息。默认的消息是：You have mail
OLDPWD	上一次的工作目录
PATH	用于查找命令的搜索路径。shell 使用这些冒号分隔的路径名来搜索需要执行的命令
PWD	当前工作目录，通过 cd 命令进行设置
PPID	父进程的进程 ID
PS1	主提示符串，默认为$



(续表)

变 量 名	作　用
PS2	次提示符串，默认为>
PS3	用于 select 命令的选择提示字符串，默认为#?
PS4	用于跟踪打开时的调试提示字符串，默认为+
RANDOM	每次引用该变量时，产生的一个随机变量
REPLY	当只读不是所提供的参数时设置
SHELL	shell 启动时，会在环境中查找这个名字。shell 把默认值赋给 PATH、PS1、PS2、MAILCHECK 和 IFS。HOME 和 MAIL 由 login 程序设置
TMOUT	指定输入等待的时间(秒)，如果在规定的时间内没有等到输入，程序退出
VISUAL	指定用于行内命令(in-line command)编辑的编辑器：emacs、gmacs 或者 vi

11.9.3 列出已设置的变量

shell 提供了 3 条行内命令来显示变量的值：set、env 和 typeset。set 命令将输出所有的变量，包括局部变量和全局变量；env 命令则只显示全局变量；typeset 命令输出所有的变量、整数、函数和已经导出的变量。set –o 命令显示 Korn shell 的所有选项。

范例 11-38

```
1   $ env       # Partial list
    LOGNAME=ellie
    TERMCAP=sun-cmd:te=\E[>4h:ti=\E[>4l:tc=sun:
    USER=ellie
    DISPLAY=:0.0
    SHELL=/bin/ksh
    HOME=/home/jody/ellie
    TERM=sun-cmd
    LD_LIBRARY_PATH=/usr/local/OW3/lib
    PWD=/home/jody/ellie/perl

2   $ typeset
    export MANPATH
    export PATH
    integer ERRNO
    export FONTPATH
    integer OPTIND
    function LINENO
    export OPENWINHOME
    export LOGNAME
    function SECONDS
    integer PPID
    PS3
    PS2
    export TERMCAP
```

```
    OPTARG
    export USER
    export DISPLAY
    function RANDOM
    export SHELL
    integer TMOUT
    integer MAILCHECK

3   $ set
    DISPLAY=:0.0
    ERRNO=10
    FCEDIT=/bin/ed
    FMHOME=/usr/local/Frame-2.1X
    FONTPATH=/usr/local/OW3/lib/fonts
    HELPPATH=/usr/local/OW3/lib/locale:/usr/local/OW3/lib/help
    HOME=/home/jody/ellie
    IFS=
    LD_LIBRARY_PATH=/usr/local/OW3/lib
    LINENO=1
    LOGNAME=ellie
    MAILCHECK=600
    MANPATH=/usr/local/OW3/share/man:/usr/local/OW3/man:/
    usr/local/man:/usr/local/doctools/man:/usr/man
    OPTIND=1
    PATH=/home/jody/ellie:/usr/local/OW3/bin:/usr/ucb:/
    usr/local/doctools/bin:/usr/bin:/usr/local:/usr/etc:/etc:/
    usr/spool/news/bin:/home/jody/ellie/bin:/usr/lo
    PID=1332
    PS1=$
    PS2=>
    PS3=#?
    PS4=+
    PWD=/home/jody/ellie/kshprog/joke
    RANDOM=4251
    SECONDS=36
    SHELL=/bin/ksh
    TERM=sun-cmd
    TERMCAP=sun-cmd:te=\E[>4h:ti=\E[>4l:tc=sun:
        TMOUT=0
    USER=ellie
    _=pwd
    name=Joe
    place=San Francisco
    x=

4   set -o
    allexport      off
    bgnice         on
    emacs          off
    errexit        off
```




```
gmacs            off
ignoreeof          off
interactive        on
keyword            off
markdirs           off
monitor            on
noexec             off
noclobber          off
noglob             off
nolog              off
nounset            off
privileged         off
restricted         off
trackall           off
verbose            off
viraw              off
xtrace             off
```

说明

1. env 命令列出所有环境变量(已导出的)。这些变量通常以大写字母命名。创建环境变量的进程把这些变量传给它的所有子进程。

2. typeset 命令显示所有的变量以及它们的属性、函数和整数值。与+选项一起使用时，typeset 命令只显示变量的名称。

3. 未指定选项时，set 命令将输出所有已设置的变量，不管它是局部变量还是导出变量(被赋为空值的变量也包括在内)。

4. set 命令与 - o 选项一起使用，可以列出所有设置为 on 或 off 的内置变量。打开这些选项时，使用加号(+)；而关闭这些选项时，则使用减号(—)。例如，set - o allexport 将打开 allexport 选项，使得所有的变量全局化。

11.9.4 复位变量

只要不被设为只读，局部变量和环境变量都可以被 unset 命令复位。

范例 11-39

```
unset name; unset TERM
```

说明

变量 name 与 TERM 在这个 shell 脚本中不再定义。

11.9.5 显示变量的值

echo 在 Korn shell 中仍然有效(主要用于 Bourne shell 和 C shell 中)。但是另外一个命令 print 有更多的选项，效率更高。两个命令都是 shell 的内置命令，print 命令的很多选项都可以控制输出的格式，这些选项如表 11-10 中所示。

表 11-10 print 命令选项

选 项	含 义
-	所有跟在它后面的参数都将不被认为是选项参数。短线允许出现连字符的参数如 - 2
- f	在 1988 年之后的版本中，用来模拟 printf
- n	标准输出不换行，如同 echo –n
- p	发送输出到连接的进程或者管道而不是标准输出上
- r	阻止 print 命令对转义序列进行解释
- R	防止 ksh 把 - 2、 - x 等作为 print 的参数。如果出现在一个参数之前，则可以省略短划线 (n 除外)
- s	输出结果作为命令被追加到历史文件中，而非标准输出中
- un	重定向输出到文件描述符 n

范例 11-38

```
1   $ print Hello my friend and neighbor!
    Hello my friend and neighbor!
2   $ print "Hello      friends"
    Hello      friends

3   $ print -r "\n"
    \n
4   $ print -s "date +%H"
    $ history -2
    132 print -s "date +%H"
    133 date +%H
    134 history -2
    $ r 133
    09
5   $ print -n $HOME
    /home/jody/ellie
6   $ var=world
    $ print ${var}wide
    worldwide
7   $ print -x is an option
    ksh: print: bad option(s)
8   $ print  - -x is an option
    -x is an option
```

说明

1．shell 解析命令行，用空格将命令行分解成多个单词(标记)，将这些单词作为参数传递给 print 命令。shell 将去除单词之间多余的空格。

2．引号表明一个单独的字符串。将该字符串作为一个单独的词，空格将被保留。

3．这是一个 raw 选项，所有的转义序列将不被解释。

4． - s 选项使得 print 命令的参数作为一个命令，附加到历史文件中。字符串“date +%H”是 print 命令的一个参数。date 字符串作为一个命令，附加在历史列表中。在执行 r



命令时被执行(历史重做命令)。

5. －n 选项取消新行。print 命令的输出结果将与 Korn shell 提示符出现在同一行。

6. 局部变量被设置为 world。大括号用于将变量名与附加在之后的字符串隔开。

7. 如果 print 函数的第一个参数是短划线开头，而接下来不也是短划线，那么该参数将被解释成它的一个选项。

8. 短划线作为一个选项，允许将一个以短划线开头的参数，作为普通字符全部显示出来。

11.9.6 转义序列

转义序列由一个反斜杠后跟一个字符构成，当该字符序列包含在引号中时，有着特殊的含义(参见表 11-11)。

如果 print 命令没有－r 与－R 选项，则利用字符串中的转义序列来格式化输出。当然该字符串必须包含在单引号或者双引号中。

范例 11-41

```
1   $ print '\t\t\tHello\n'
                Hello

    $

2   $ print "\aTea \tTime!\n\n"
    Ding ( bell rings ) Tea      Time!
```

说明

1. 转义序列必须包含在单引号或者双引号中。转义序列\t 表示一个制表符；\n 表示换行。Hello 之后有 3 个制表字符和一个新行。

2. 转义序列\a 产生 1 个响铃(\07)。\t 产生 1 个制表符。两个\n 表示两次换行。

表 11-11 转 义 序 列

转义序列	含义
\a	响铃字符
\b	退格
\c	取消新行并忽略其后的任何参数
\f	换页
\n	换行
\r	回车
\t	制表符
\v	纵向制表符
\\	反斜杠
\0x	一个用 1、2 或者 3 位 ASCII 值表示的 8 位字符，如\0124
\E	作为转义序列使用，仅出现在 1988 年之后的版本中

11.9.7 变量表达式和扩展修饰符

可以用一些特殊的修饰符来测试和修改变量表达式。修饰符首先提供一个简捷的条件测试，用来检查某个变量是否已经被设置，然后根据测试结果给变量赋一个默认值。这些表达式可以和条件语句如 if、elif 一起使用。请参见表 11-12 中列出的变量修饰符。

表 11-12 变量修饰符

表达式	功能
${variable: - word}	如果 variable 已被设置且值非空，就取其值。否则，取 word 的值
${variable:=word}	如果 variable 已被设置且值非空，则将它的值设置为 word。variable 的值将被永久替换。而位置参数不能用这种方式赋值
${variable:+word}	如果 variable 已被设置且值非空，则替换为 word。否则，什么都不代入(代入空值)
${variable:?word}	如果 variable 已被设置且值非空，就代入它的值。否则，显示 word 并且从 shell 中退出。如果省略了 word，就会显示信息：parameter null or not set

冒号对修饰符来说是可选的。和冒号配合使用时，修饰符(-、=、+、?)将检查变量是否尚未赋值或值为空。不加冒号时，值为空的变量也被认为已设置。

范例 11-42

(使用临时默认值)

```
1   $ fruit=peach
2   $ print ${fruit:-plum}
    peach
3   $ print ${newfruit:-apple}
    apple
4   $ print $newfruit

5   $ print ${TERM:-vt120}
    sun-cmd
```

说明

1. 将变量 fruit 赋值为 peach。

2. 这个特殊的修饰符将检查变量 fruit 是否已设置。如果已设置，就显示 peach。否则，用 plum 替换 fruit，并显示。

3. 变量 newfruit 未被设置。apple 的值将被显示。

4. 变量 newfruit 未被设置，所以不会显示任何东西。在第 3 行中，表达式将被简单地替换为单词 apple，并显示出来。

5. 变量 TERM 的值未被设置，将显示其默认的值 vt120。在这个示例中，终端已经被设置为 sun-cmd，即一种 Sun 工作站。



范例 11-43
(赋值永久默认值)

```
1  $ name=
2  $ print ${name:=Patty}
   Patty
3  $ print $name
   Patty
4  $ print ${TERM:=vt120}
   vt120
   $ print $TERM
   vt120
```

说明

1. 赋给变量 name 一个空值。

2. 特殊修饰符“:=”将检查变量 name 是否尚未被设置。如果已被设置，就不会被改变；如果尚未设置或值为空，就将等号右边的值赋给它。由于之前已将变量 name 设置为空，所以现在要把 Patty 赋给它。这个设置将是永久的。

3. 变量 name 的值仍然为 Patty。

4. 如果变量 TERM 没有被设置，那么它将被永久地设置为 vt120。

范例 11-44
(临时替换赋值)

```
1  $ foo=grapes
2  $ print ${foo:+pears}
   pears
   $ print $foo
   grapes
```

说明

1. 将变量 foo 的值设置为 grapes。

2. 特殊的修饰符“:+”将检查变量 name 是否已被设置。如果已被设置，就用 pears 临时替换 foo，否则，返回空。变量 foo 还是原来的值。

范例 11-45
(基于默认值生成错误消息)

```
1  $ print ${namex:?"namex is undefined"}
   ksh: namex: namex is undefined
2  $ print ${y?}
   ksh: y: parameter null or not set
```

说明

1. 修饰符“:?”检查变量是否已被设置。如果尚未设置，就把问号右边的信息显示在标准错误输出上，且在变量名后。如果此时在执行了脚本，将退出脚本。

2. 如果问号后面没有提供报错信息，Korn shell 就向标准错误输出发送默认的消息。若不加冒号，则问号修饰符将空变量看成是已设置的而不显示出错信息。

范例 11-46

```
(系统脚本中的一行)
if [ "${uid:=0}" -ne 0 ]
```

说明

如果 UID(用户 ID)有值，那么它不会被更改。如果没有值，那么它会被赋为 0(超级用户)。然后再来判断该变量是否为 0。这行程序是从/etc/shutdown 程序中取出(SVR4/Solaris 2.5)的。此处给出的是如何使用变量修饰符的示例。

11.9.8 变量子字符串扩展

使用模式匹配参数，可以从目标字符串中的头、尾处开始，取出特定位置的子字符串。这类操作符在将特定的路径名元素从一个目录的头、尾中去除时，使用得最为普遍。参见表 11-13。

表 11-13 变量子字符串扩展

表 达 式	功 能
${variable%pattern}	从尾部开始在变量 variable 中寻找 pattern 最小的匹配部分，并将其去除
${variable%%pattern }	从尾部开始在变量 variable 中寻找 pattern 最大的匹配部分，并将其去除
${variable#pattern}	从头部开始在变量 variable 中寻找 pattern 最小的匹配部分，并将其去除
${variable##pattern}	从头部开始在变量 variable 中寻找 pattern 最大的匹配部分，并将其去除

范例 11-47

```
1   $ pathname="/usr/bin/local/bin"
2   $ print ${pathname%/bin*}
    /usr/bin/local
```

说明

1. 局部变量 pathname 被赋为/usr/bin/local/bin。

2. %操作符从尾部开始在 pathname 中寻找最小的后面有零个或多个字符的/bin 匹配部分，在这里找到的是/bin，并将它去除。

范例 11-48

```
1   $ pathname="usr/bin/local/bin"
2   $ print ${pathname%%/bin*}
    /usr
```

说明

1. 局部变量 pathname 被赋值为/usr/bin/local/bin。

2. %%操作符从尾部开始在/usr/bin/local/bin 中寻找最大的后面有零个或多个字符的/bin 匹配部分，在这里找到的是/bin/local/bin，并将它去除。

范例 11-49

```
1   $ pathname=/home/lilliput/jake/.cshrc
```



```
2  $ print ${pathname#/home}
   /lilliput/jake/.cshrc
```

说明

1. 局部变量 pathname 被赋值为/home/lilliput/jake/.cshrc。

2. #操作符从头部开始在/home/lilliput/jake/.cshrc 中寻找最小的/home 匹配部分，在这里找到的是/home，并将它去除。

范例 11-50

```
1  $ pathname=/home/liliput/jake/.cshrc
2  $ print ${pathname##*/}
   .cshrc
```

说明

1. 局部变量 pathname 被赋值为/home/lilliput/jake/.cshrc。

2. ##操作符从头部开始在/home/lilliput/jake/.cshrc 中寻找最大的包含零个或多个字符且最后是一个斜杠的匹配部分，在这里找到的是/home/lilliput/jake/，并将它去除。

11.9.9 变量属性：typeset 命令

typeset 可以控制一些变量属性，例如：大小写、宽度、左端对齐或者右端对齐等。在 typeset 改变这些变量的属性时，修改将被永久保存。typeset 除了具备以上功能外，还有其他一些功能，参见表 11-14。

表 11-14 typeset 命令的其他用法

命　令	所做的动作
typeset	显示所有的变量
typeset –i num	设置变量只能接受整数值
typeset –x	显示所有已导出的变量
typeset a b c	如果在一个函数中定义，就创建局部变量 a、b 和 c
typeset –r x=foo	将变量 x 赋为 foo，然后设置为只读

范例 11-51

```
1  $ typeset -u name="john doe"
   $ print "$name"
   JOHN DOE            # Changes all characters to uppercase.

2  $ typeset -l name
   $ print $name
   john doe            # Changes all characters to lowercase.

3  $ typeset -L4 name
   $ print $name
   john                # Left-justified fixed-width 4-character field.
```

```
4   $ typeset -R2 name
    $ print $name       # Right-justified fixed-width 2-character field.
    hn

5   $ name="John Doe"
    $ typeset -Z15 name        # Null-padded sting, 15-space field width
    $ print "$name"
     John Doe

6   $ typeset -LZ15 name       # Left-justified, 15-space field width.
    $ print "$name$name"
    John Doe      John Doe

7   $ integer n=25
    $ typeset -Z15 n            # Left-justified, zero-padded integer.
    $ print "$n"
    000000000000025

8   $ typeset -lL1 answer=Yes   # Left justify one lowercase letter.
    $ print $answer
    y
```

说明

1. typeset 命令的 - u 选项将变量中的所有字符转变成大写。

2. typeset 命令的 - l 选项将变量中的所有字符转变成小写。

3. typeset 命令的 - L 选项将变量 name 转变成一个左端对齐且包含 4 个字符的字符串 john。

4. typeset 命令的 - R 选项将变量 name 转变成一个右端对齐且包含 2 个字符的字符串 hn。

5. 变量 name 被赋值为 John Doe。typeset 命令的 - Z 选项将变量 name 转变成长度为 15 的用 null 补齐的字符串。为了保留空格，需将变量包含在引号中。

6. 变量 name 被转变成一个左端对齐的、长度为 15、null 补齐的字符串。

7. 变量 n 是一个值为 25 的整数(见 typeset -i，如表 11-14 所示)。typeset 命令将整数 n 转变成一个用 0 填充、长度为 15、左端对齐的数值。

8. 变量 answer 被赋值为 Yes，并且被转变成小写形式、左端对齐且包含一个字符的字符串。这个功能在脚本中处理用户输入时非常有用。

11.9.10 位置参数

shell 脚本通常使用特殊内置变量——位置参数(positional parameter)从命令行接受参数，位置参数还被函数用来保存传给它的参数，参见表 11-15。这些变量之所以被称为位置参数，是因为 shell 用它们在参数列表中的位置来指代它们，即 1、2、3 等。



表11-15 位置参数

参数表达式	作　用
$0	表示当前shell脚本的名称
$1 - $9	代表第1到第9个位置参数
${10}	第10个位置参数
$#	其值为位置参数的个数
$*	其值为所有的位置参数
$@	除了加双引号的情况，作用与$*相同
"$*"	其值为"$1 $2 $3"等
"$@"	其值为"$1""$2""$3"等

shell脚本的名称保存在变量$0中。可以使用set命令来设置或重置位置参数。

范例 11-52

```
1   $ set tim bill ann fred
    $ print $*          # Prints all the positional parameters.
    tim bill ann fred

2   $ print $1          # Prints the first position.
    tim

3   $ print $2 $3        # Prints the second and third position.
    bill ann

4   $ print $#           # Prints the total number of positional parameters.
    4

5   $ set a b c d e f g h i j k l m
    $ print $10     # Prints the first positional parameter followed by a 0.
    a0

    $ print ${10} ${11}  # Prints the 10th and 11th positions.
    j k

6   $ print $#
    13

7   $ print $*
    a b c d e f g h i j k l m

8   $ set file1 file2 file3
    $ print \$$#
    $3

9   $ eval print \$$#
    file3
```

说明

1. set 命令给位置参数赋值。特殊变量$*包含所有已设置的位置参数。
2. 显示第 1 个位置参数 tim 的值。
3. 显示第 2 个和第 3 个位置参数的值，即 bill 和 ann 的值。
4. 特殊变量$#的值是当前已设置的位置参数的个数。
5. set 命令重置所有的位置参数，原来的位置参数列表被清除。如果位置参数的下标超过 9，则必须用大括号括起来。否则就会看成是取第一个位置参数的值，后面跟一个数字 0。
6. 目前位置参数的个数为 13。
7. 显示所有位置参数的值。
8. 把位置参数重置为 file1、file2 和 file3。前一个美元符被转义，后面的$#代表参数的个数。echo 命令输出为$3。
9. 执行命令之前，eval 命令对命令行进行第二次解析。第一次解析时，shell 把\$$#替换为$3。第二次解析时，shell 又将$3 替换为它的值，即 file3。

11.9.11 其他特殊变量

Korn shell 有一些特殊的内置变量。如表 11-16 所示。

表 11-16 特殊变量

变 量	含 义
$$	当前 shell 的 PID
$ -	当前的 ksh 选项设置
$?	shell 执行的上一条命令的退出状态
$!	最近一个进入后台的作业的 PID

范例 11-53

```
1   $ print The pid of this shell is $$
    The pid of this shell is 4725
2   $ print The options for this korn shell are $-
    The options for this korn shell are ismh
3   $ grep dodo /etc/passwd
    $ print $?
    1
4   $ sleep 25&
    [1]    400
    $ print $!'
    400
```

说明

1. 变量$$中保存的是本进程的 PID。
2. 变量$-列出交互式 korn shell 的所有选项。
3. grep 命令在/etc/passwd 文件中搜索字符串 dodo，变量？包含上一条执行完成的命



令的退出状态，因为 grep 命令的返回状态值为 1，可以判断出查找失败。若查找成功，则退出状态值为 0。

4. sleep 命令后的&符号使得这条命令在后台执行，$!变量包含上一条以后台方式执行命令的 PID。

11.10 引用

引号用于防止特殊的元字符被解释。在所有 shell 脚本中这都是导致调试困难的原因。单引号必须配对使用，它可以避免特殊元字符被 shell 解释。双引号也必须配对，它也可以避免大多数特殊元字符被 shell 解释，但是它允许处理变量和命令替换字符。双引号将保护其中的单引号，单引号保护其中的双引号。与 Bourne shell 不同，如果检查到不配对的引号，Korn shell 会向标准错误输出发送错误信息以及未匹配的行。

11.10.1 反斜杠

反斜线用于防止单个字符被解释。

范例 11-54

```
1   $ print Where are you going\?
    Where are you going?
2   $ print Start on this line and \
    > go to the next line.
    Start on this line and go to the next line.
```

说明

1. 特殊元字符 ? 被反斜线保护，使之不被 shell 解释。
2. 换行符不会被解释，后一行将成为前一行的一部分，> 符号是 Korn shell 的次提示符。

11.10.2 单引号

单引号必须配对使用，以用来保护所有元字符不被解释。要显示一个单引号，必须用双引号把它包含起来，或者用一个反斜杠对其进行转义。

范例 11-55

```
1   $ print 'hi there
    > how are you?
    > When will this end?
    > When the quote is matched
    > oh'
    hi there
    how are you?
    When will this end?
    When the quote is matched
    oh
```

```
2   $ print 'Do you need $5.00?'
    Do you need $5.00?

3   $ print 'Mother yelled, "Time to eat!"'
    Mother yelled, "Time to eat!"
```

说明

1. 因单引号在这一行中没有配对，Korn shell 产生次提示符，直到引号配对。
2. 单引号保护所有元字符不被解释，本例中 $ 和 ? 被当作普通字符使用。
3. 单引号保护字符串中的双引号不被解释，双引号被当作普通字符使用。

11.10.3 双引号

双引号必须配对使用，它允许其中的变量和命令替换但保护其他元字符不被 Shell 解释。

范例 11-56

```
1   $ name=Jody
2   $ print "Hi $name, I'm glad to meet you!"
    Hi Jody, I'm glad to meet you!

3   $ print "Hey $name, the time is `date`"
    Hey Jody, the time is Fri Dec 18 14:04:11 PST 2004
```

说明

1. 变量 name 被赋值为 Jody。
2. 双引号阻止了除$之外的元字符被 shell 解释。双引号中执行了变量替换。
3. 当使用双引号时，变量和命令替换将同时进行：$name 被扩展，反引号中的命令 date 也被执行。

11.11 命令替换

命令替换用来将一条命令的输出赋给某个变量，或替换为一个字符串。Bourne shell 和 C shell 使用反引号实现命令替换，而 Korn shell 也允许使用这种已“弃用”的格式[4]，使用圆括号是推荐的方法，因为它具有较简单的引用规则，从而使得嵌套命令容易使用。

格式：

```
`Unix command`      # Old method with backquotes
$ (Unix command)    # New method
```

范例 11-57

(旧方法)

```
1   $ print "The hour is `date +%H`"
```

④ 使用反引号是 Bourne shell 和 C shell 中进行命令替换的一种旧格式，但是语法仍然合法。在本节中将介绍 Korn shell 引入的一种新方法。




```
    The hour is 09
2   $ name=`nawk -F: '{print $1}' database`
    $ print $name
    Ebenezer Scrooge
3   $ ls `ls /etc`
    shutdown
4   $ set `date`
5   $ print $*
    Sat Oct 13 09:35:21 PDT 2004
6   $ print $2 $6
    Oct 2004
```

说明

1. 命令 date 的输出被替换到字符串中。

2. 命令 nawk 的输出被赋给变量 name，并显示出来。

3. 反引号中 ls 命令的输出，是/etc 目录中的所有文件的列表，这些文件名成为第一个 ls 命令的参数，这样，和/etc 目录中文件同名称的文件将被列出来。

4. set 命令把 date 命令的输出赋给位置参数，空白符将这些词语分隔成不同的参数。

5. 变量$*包含所有的参数，date 命令的结果被保存在$*变量中，且每个参数用空白符分隔。

6. 显示第 2 个参数和第 6 个参数。

在命令替换中使用反引号的方法在范例 11-58 中进行了说明。

范例 11-58

```
(ksh 的新方法)
1   $ d=$(date)
    $ print $d
    Sat Oct 20 09:35:21 PDT 2004
2   $ line = $(< filex)
3   $ print The time is $(date +%H)
    The time is 09
4   $ machine=$(uname -n)
    $ print $machine
    jody
5   $ dirname="$(basename $(pwd))"      # Nesting commands
    $ print $dirname
    bin
```

说明

1. date 命令被圆括号包围，该命令的输出被赋给变量 d，并显示出来。

2. 从文件中得到的输入被赋给变量 line，符号< filex 和命令 cat filex 等效。当圆括号前是一个$符号时，将对括号中的字符串执行命令替换。

3. UNIX 的 date 命令和它的小时参数“+%H”被圆括号包围。执行命令替换，将结果替换到 print 字符串中。

4. 执行命令替换，UNIX uname 命令的结果将被赋给变量 machine。

5．将变量 dirname 赋值为当前工作目录名(不含路径)，进行命令替换嵌套。首先执行 pwd 命令获得当前工作目录的完整路径，然后以此为参数传递给 UNIX 命令 basename。basename 命令将截取路径名的最后一段之外的所有字符。反引号中不允许出现命令嵌套。

11.12　函数

本节介绍函数，以便读者能以交互方式使用函数，或者把它们保存在初始化文件中。在后面讨论脚本时，我们将作更深入的介绍。当使用别名无法满足需要时，就可以使用函数，例如经常需要传递参数。函数经常定义在用户初始化文件.profile 中，它们类似于一个小型脚本，但是和脚本不同的是，函数运行在当前环境中。也就是说，shell 不会产生一个子进程来执行函数。所有调用函数的 shell 将共享变量。函数经常用于改善脚本的模块化，一经定义，它们可以反复被调用，甚至也可以存放在另一个目录中。

函数在调用前必须先定义，有两种格式可以定义函数：一种是 Bourne shell 的格式，另一种是 Korn shell 的新格式。typeset 和 unset 命令可以分别用来列出当前定义的函数和取消一个函数的定义。参见表 11-17。

表 11-17　用于列出和设置函数的命令

命　令	功　能
typeset –f	列出函数及其定义，函数其实是它的别名
typeset +f	列出函数名称
unset –f name	取消函数的定义

11.12.1　函数的定义

函数可以用两种格式来定义：Bourne shell 格式(依然可用，向上兼容)和新的 Korn shell 格式。函数在使用前必须先定义[5]。

格式

```
(Bourne Shell)
函数() {命令;命令;}
(Ksh)
函数{命令;命令; }
```

范例 11-59

```
1   $ function fun { pwd; ls; date; }

2   $ fun
    /home/jody/ellie/prac
    abc      abc123    file1.bak  none      nothing  tmp
```

[5] POSIX 标准定义了使用 Bourne shell 定义函数的语法，但是与新的 Korn shell 定义一样，变量和自陷(trap)在作用域上不是局部的。




```
    abc1    abc2    file2      nonsense  nowhere  touch
    abc122  file1   file2.bak  noone     one
    Mon Feb 9 11:15:48 PST 2004

3   $ function greet { print "Hi $1 and $2"; }

4   $ greet tom joe             # Here $1 is tom and $2 is joe
    Hi tom and joe

5   $ set jane nina lizzy
6   $ print $*
    jane nina lizzy

7   $ greet tom joe
    Hi tom and joe

8   $ print $1 $2
    jane nina
```

说明

1. 命名并定义函数 fun，在函数名之后可以跟随一系列被花括号包围的语句，每个语句用分号隔开，第一个花括号后必须有一个空格，否则会导致语法错误(ksh:syntax error:'}' unexpected)。函数在使用之前必须先定义。

2. 调用函数类似于调用脚本和别名，函数中定义的所有语句会被依次执行。

3. 在函数 greet 中用到了两个位置参数，当在函数名后跟随参数时，参数的值就会被赋给位置参数。

4. 函数的参数 tom 和 joe 分别被赋给$1 和$2，函数内部的参数是私有的，不会对函数外部产生影响。

5. 位置参数在命令行中设置，它们和函数内部定义的参数没有关系。

6. $*显示的是当前设置的位置参数的值。

7. 调用函数 greet，其中位置参数$1 和$2 分别被赋值为 tom 和 joe。

8. 在命令行中赋值的位置变量，其值没有被函数中的赋值所影响。

11.12.2 函数和别名

在处理命令行时，shell 首先查找别名，其次查找内置命令，最后查找函数。如果一个函数与一个内部命令同名，则将优先执行内置命令。可以为内置命令定义别名，然后用别名来命名函数，这样可以改变处理的顺序。

范例 11-60

```
(ENV 文件)
1   alias cd=_cd
2   function _cd {
3   \cd $1
4   print $(basename $PWD)
5   }
```

```
(命令行)
$ cd /
/
$ cd $HOME/bin
bin
$ cd ..
ellie
```

说明

1. cd 的别名设置为_cd。
2. 定义函数_cd。左花括号表示函数定义开始。
3. 如果在别名前有一个反斜杠，则不执行别名替换。本行中反斜杠后跟 cd，表示执行的是内置命令 cd，而不是别名 cd。如果没有反斜杠，函数将产生递归，shell 将显示出错信息：cd_:recursion too deep。$1 是传递给 cd 的参数(目录名)。
4. 显示目录名(而不是完整路径)。
5. 右花括号标志函数定义结束。

11.12.3 列出函数

使用 typeset 命令来列出函数及其定义。

范例 11-61

```
(命令行)
1  $ typeset -f
   function fun
   {
   pwd; ls; date; }
   function greet
   {
   print "hi $1 and $2"; }
2  $ typeset +f
   fun
   greet
```

说明

1. 带 - f 选项的 typeset 命令，列出函数及其定义。
2. 带 +f 选项的 typeset 命令，仅列出已定义函数的名称。

11.12.4 取消函数的定义

当一个函数被 unset 命令取消时，它将从 shell 的内存中清除。

范例 11-62

```
(命令行)
1  $ typeset -f
   function fun
   {
      pwd; ls; date; }
   function greet
```




```
   {
   print "hi $1 and $2"; }

2  $ unset -f fun
3  $ typeset -f
   function greet
   {
   print "hi $1 and $2"; }
```

说明

1. typeset –f 命令显示函数及其定义。在此显示两个函数，fun 和 greet。
2. 带 –f 选项的 unset 命令，取消 fun 函数的定义，并将它从 shell 内存中清除。
3. 现在执行 typeset –f 命令时，将不再显示函数 fun，而只显示函数 greet。

11.13 标准 I/O 和重定向

每当一个程序启动时，shell 会将打开 3 个文件(称为流)：标准输入 stdin、标准输出 stdout 和标准错误 stderr。标准输入一般来自键盘输入，和文件描述符 0 关联；标准输出一般是显示屏，同文件描述符 1 关联；标准错误一般也显示在显示屏上，与文件描述符 2 关联。这 3 个标准 I/O 可以重定向到文件，参见表 11-18。

表 11-18 重 定 向

操 作 符	功 能
<file	从 file 重定向输入
>file	重定向输出到 file
>>file	重定向并追加输出到 file
2>file	重定向错误输出到 file
2>>file	重定向并追加错误输出到 file
1>&2	重定向标准输出到错误输出
2>&1	重定向错误输出到标准输出

范例 11-63

(命令行)

```
1  $ tr '[A-Z]' '[a-z]' < myfile   # Redirect input
2  $ ls > lsfile            # Redirect output
   $ cat lsfile
   dir1
   dir2
   file1
   file2
   file3
3  $ date >> lsfile         # Redirect and append output
   $ cat lsfile
   dir1
```

```
   dir2
   file1
   file2
   file3
   Mon Sept 20 12:57:22 PDT 2004
4  $ cc prog.c 2> errfile          # Redirect error
5  $ find . -name \*.c -print > founditfile 2> /dev/null
6  $ find . -name \*.c -print > foundit 2>&1
7  $ print "File needs an argument" 1>&2
8  $ function usage { print "Usage: $0 [-y] [-g] filename" 1>&2 ; exit 1; }
```

说明

1. 标准输入从文件 myfile 重定向为 UNIX 命令 tr。所有大写字母均转换成小写字母。
2. ls 命令的输出被重定向到文件 lsfile。
3. date 命令的输出被重定向并追加到文件 lsfile 的尾部。
4. 程序 prog.c 被编译，如果编译失败，错误被定向到 errfile 文件。
5. find 命令在当前工作目录中查找所有文件名以.c 结尾的文件，并把这些文件名保存在 founditfile 中。find 命令执行的错误被送到/dev/null。
6. find 命令在当前工作目录中查找所有文件名以.c 结尾的文件，标准输出(文件描述符 1)重定向到文件 foundit，标准错误输出(文件描述符 2)也被送到标准输出的重定向文件 foundit。
7. print 命令的输出被送到标准错误输出。标准输出被转移到标准错误输出，也就是说标准输出被重定向到标准错误输出的位置，这里为终端显示屏。这样可以把错误信息和正常输出分离。
8. 定义函数 usage，当它被调用时，会输出一行用法提示，把标准输出定向到标准错误输出，然后退出。这样的函数经常在脚本中使用。

11.13.1 exec 命令和重定向

exec 命令可用于把当前运行的程序用将要运行的程序取代，它的另一个作用是在不创建子 shell 的情况下改变标准输入或标准输出。如果使用 exec 命令打开一个文件，随后的每一次读操作都会把文件指针向后移一行，直到文件尾。文件必须被关闭之后重新打开才能从文件的头部开始读。但是如果使用 UNIX 工具程序如 cat 和 sort，操作系统就会在每一条命令完成后关闭文件。Exec 命令的有关功能请参见表 11-19。

表 11-19 exec 命 令

命 令	功 能
exec ls	ls 将代替 shell 执行，执行完该命令后，不会返回启动 ls 命令的 shell
exec <filea	打开 filea 作为标准读输入
exec >filex	打开 filex 作为标准写输出
exec 2>errors	打开 errors 作为标准错误写输出
exec 2>>errors	打开 errors 作为标准错误写输出和追加输出
exec 2>/dev/console	发送所有错误消息到 console
exec 3<datfile	打开 datfile 作为读输入，文件描述符为 3



(续表)

命　令	功　能
sort <&3	对 datefile 文件排序
exec 4>newfile	打开 newfile 用于写，文件描述符为 4
ls >&4	ls 的输出重定向到 newfile
exec 5<&4	把文件描述符 4 复制到文件描述符 5，它们都指向 newfile
exec 3<& -	关闭文件描述符 3，datafile

11.13.2 重定向与子 shell

当一个命令的输出从屏幕重定向到文件时，Korn shell 会创建一个子 shell 来重新分配文件描述符。参见图 11-2。

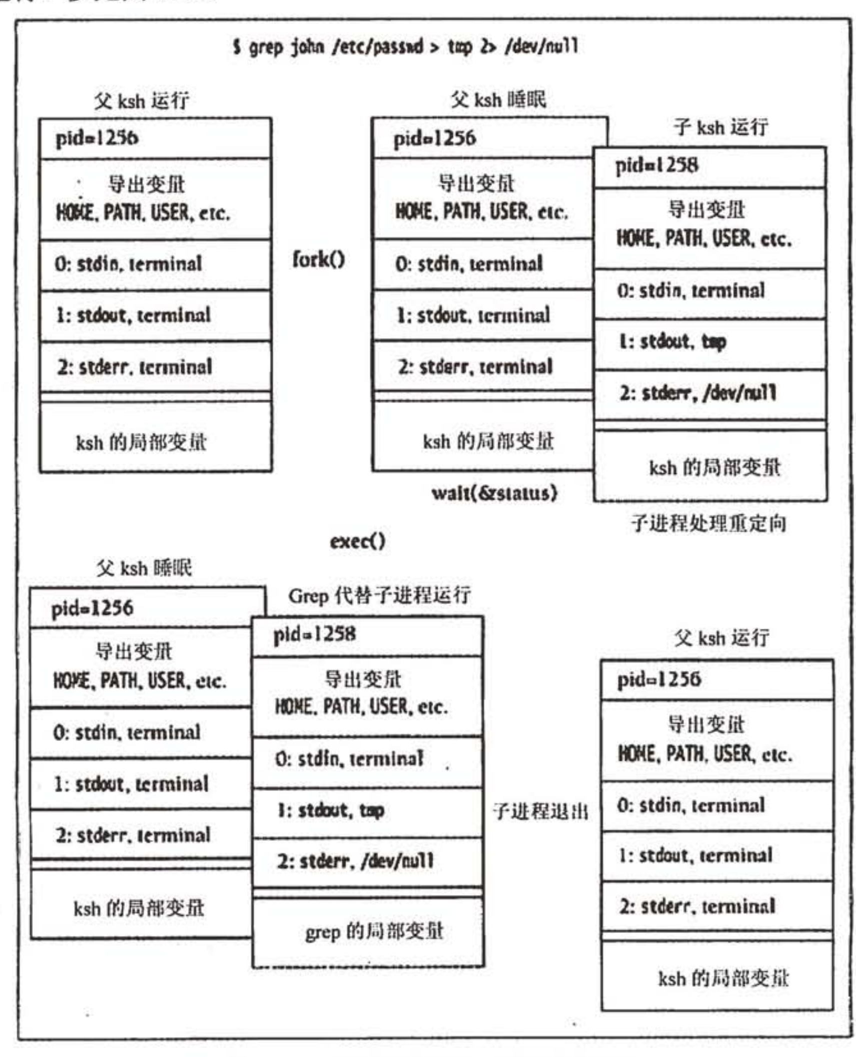


图 11-2　重定向标准输出和标准错误输出

11.14 管道

管道接收管道符号左边的命令的输出，并把它作为管道符号右边命令的输入，一条管道线可以包含多个管道。

范例 11-64

```
1   $ who > tmp
2   $ wc -l tmp
    4 tmp
3   $ rm tmp
4   $ who | wc -l       # Using the pipe
```

说明

第 1 行到第 3 行用来统计有多少用户登录(who)，并把命令输出保存到文件 tmp 中，使用 wc –l 可以计算 tmp 文件的行数，最后删除 tmp 文件。这样就得到已登录用户的数量。管道在一条命令中完成相同的功能。

1. who 命令的输出被重定向到 tmp 文件。
2. wc –l 命令显示 tmp 文件中的行数。
3. 删除 tmp 文件。
4. 使用管道工具，用一个步骤就能完成上述 3 个步骤的功能：who 命令的输出被送到一个匿名内核缓冲区(代替了需要占用磁盘空间的临时文件)，wc –l 命令从该缓冲区读取输入，并将命令的输出发送到屏幕上。

范例 11-65

```
1   $ ls | more
    < lists (ls) all files one page at a time (more) >
2   $ du ~ | sort -n | sed -n '$p'
    72388  /home/jody/ellie
3   $ cat | lp or  cat | lpr
```

说明

1. ls 命令的输出通过管道送给 more 命令，more 命令接受输入。输出每次显示一页。
2. du(disk usage，磁盘已使用空间)命令的输出将按数字排序，通过管道送到 sed(stream editor，流编辑器)命令，sed 命令只显示最后一行($p)。
3. cat 命令从标准输入读信息，它的输出被管道送到打印机(SVR4 是 lp，BSD 是 lpr)。

here 文档和重定向输入

一个 here 文档为一些程序(如 mail、sort、cat)截取输入，这些输入位于两个相同的词或符号之间。第一个词跟在 UNIX 命令和 << 符号之后，以后的行包含将由命令接收的输入，最后一行将出现与第一个词完全匹配的词，除了这个词以外没有别的输入，这个词被称为终止符，标志着输入的结束。这和用 Ctrl+D 组合键来结束输入是完全一样的。终止符



周围不能有空格；如果第一个词是跟在<< - 符号的后面，那么终止符的前面可以有 tab 键(也只能有 tab 键)。通常，here 文档用于 shell 脚本编写但不能交互使用，一个常见的用处是为脚本生成一个菜单。

格式：

```
Unix 命令 << 终端
            输入行
            ......
            终端
```

范例 11-66

(命令行)

```
1   $ cat << FINISH
2   > Hello there $LOGNAME
3   > The time is $(date)
    > I can't wait to see you!!!
4   > FINISH
5   Hello there ellie
    The time is Sun May 30 19:42:16 PDT 2004
    I can't wait to see you!!
6   $
```

说明

1. UNIX/Linux cat 程序接受输入，直到 FINISH 在某行单独出现。FINISH 是一个用户自定义的终止符。
2. 在 here 文档中执行变量替换，> 符号是 Korn shell 的次提示符。
3. 在 here 文档中执行命令替换。
4. 用户定义的终止符 FINISH，标志着 cat 程序的输入结束。FINISH 前后不能有空格，必须是自成一行。
5. 显示 cat 程序的输出。
6. 回到 shell 提示符。

范例 11-67

(来源于.profile 文件)

```
1   print "Select a terminal type"
2   cat << EOF
      [1] sun
      [2] ansi
      [3] wyse50
3   EOF
4   read TERM
    ...
```

说明

1. 提示用户选择终端类型。
2. 菜单显示在屏幕上。这是一个 here 文档，意味着从这里开始直到下一个匹配的 EOF

符号(在行 3 处)，所有的输入都被 cat 命令接收。您可以用一系列 echo 命令来达到相同效果，但是从视觉上看这样好得多。

3. EOF 是用户定义的终止符，标志着 here 文档结束。它必须紧靠左边界，且前后无空格。

4. 用户从键盘的输入会被读取并赋给变量 TERM。

范例 11-68

```
(命令行)
1   $ cat <<- DONE
    >Hello there
    >What's up?
    >Bye now The time is $(date).

2   >     DONE

3   Hello there
    What's up?
    Bye now The time is Sun May 30 19:48:23 PDT 2004.
```

说明

1. cat 命令接受输入，直到自成一行的 DONE 出现。<<- 符号允许终止符前有一个或多个 tab(>是 shell 的次提示符)。

2. 匹配的 DONE 终止符之前有一个制表符。从行 1 的第一个 DONE 到本行的第二个 DONE 之间的文本，都将作为输入传递给命令 cat。

3. cat 命令的输出显示在屏幕上。

11.15 time 命令

11.15.1 time 命令

time 命令是 ksh 的内置命令，它显示以下信息到标准错误输出：执行一条命令的总时间、用户时间和系统时间。

范例 11-69

```
1   $ time sleep 3
    real  0m3.15s   took 3.15 seconds to run
    user  0m0.01s   sleep used its own code for .01 seconds
    sys   0m0.08s   and kernel code for .08 seconds

2   $ time ps -ef | wc -l     # time is measured for all commands in the pipeline
    38
    real  0m1.03s
    user  0m0.01s
    sys   0m0.10s
```



说明

1. time 命令显示执行该 sleep 命令所用的总时间、用户部分所用的时间和内核所用的时间。该 sleep 命令的运行用了 3.15 秒。

2. 显示 ps 命令和 wc 命令所用的时间。

11.15.2 TMOUT 变量

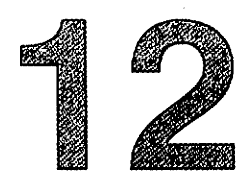
TMOUT 变量是一个整型变量，它可以用于设置一个时间段，用户必须在这段时间内输入命令。TMOUT 的默认值为 0，这允许用户在出现 PS1 提示符后的任意时间段内进行输入。如果 TMOUT 被设置为一个大于 0 的值，shell 将在该值规定的时间到期后退出。在 shell 退出前，还将分配 60 秒作为提示时间。

范例 11-70

```
$ TMOUT=600
time out in 60 seconds due to inactivity
ksh: timed out waiting for input
```

说明

变量 TMOUT 被设置为 600 秒。如果用户在 600 秒内没有做任何事情，屏幕上将显示一条消息，再过 60 秒后，shell 将退出。如果您仍不输入任何内容，则当前 shell 60 秒后退出。





Korn shell 编程

12.1 简介

当命令是通过一个文件执行，而不是在命令行中执行时，则称该文件为 shell 脚本，并且 shell 将以非交互的方式运行。编写 Korn shell 脚本的过程有一些步骤，在下面的章节中我们将对此进行阐述。

创建一个 shell 脚本的步骤

shell 脚本通常是在编辑器中编写，由命令和注释组成，注释是跟在井号(#)后面的内容。

第一行 位于脚本左上角的第一行会指出要用哪个程序来执行脚本中的行。这一行通常写成：

```
#!/bin/ksh
```

这一行必须是脚本顶端第一行，其中 #! 被称为幻数，内核根据它来确定该用哪个程序来翻译脚本中的行。Korn shell 还提供一些符号选项，用来控制 shell 的行为，这些选项将在本章的最后表 12-16 中列出。

注释 注释是跟在井号(#)后的行。注释可以自成一行，也可以跟在脚本命令后面与命令共处一行。注释被用来对脚本作注解。有时候，如果没有注释，就很难理解脚本究竟可以用来做什么。尽管注释很重要，但是脚本中却经常缺少注释，甚至根本就没有注释。我们要尽量养成为所做的工作写注释的习惯，不光方便别人，也方便自己。

可执行语句与 Kon shell 结构 Korn shell 程序由 UNIX 命令、Korn shell 命令、编程结构和注释组成。

命名和存储脚本 脚本文件的名称，最好是一个有意义的名字，同时不要与其他的 UNIX 命令或别名冲突。例如，你可能想把脚本命名为 test，因为它是一个只执行一些简单测试的过程，但 test 是一个内置的命令，一旦执行就会发现执行的并不是所写的 test 脚本。另外，如果你把脚本命名为 foo、goo、boobar 等很随意的名字，几天甚至几小时后你就可



能搞不清脚本的用途。

在测试完脚本并确认它没有错误后，把它保存到一个目录中，然后设置好搜索路径，这样在任何目录层次下都可以执行这个脚本。

范例 12-1

```
1   $ mkdir ~/bin
2   $ mv myscript ~/bin
(在.profile 中)
3   export PATH=${PATH}:~/bin
4   $ . .profile
```

说明

1. 通常把脚本保存在用户主目录的 bin 目录中。
2. 脚本 myscript 被移动到目录 bin 中。
3. 在.profile 文件中修改 PATH 变量，把 bin 目录加入到变量中。
4. 命令“.”将使文件.profile 在当前环境下执行，这样不必登出再登入系统就可以使新的设置生效。

使脚本可执行　当您创建文件时，并没有自动授予文件的执行权限(在不考虑 umask 设置的情况下)。如果要运行脚本，就必须给它执行权限。可以用 chmod 命令来打开脚本的执行权限。

范例 12-2

```
1   $ chmod +x myscript
2   $ ls -lF myscript
    -rwxr--xr--x  1 ellie     0 Jul 12 13:00 myscript*
```

说明

1. chmod 命令为用户、组及其他用户、其他组打开执行权限。
2. ls 命令打印的信息显示，所有用户对 joker 文件都有执行权限。文件名末尾的星号表示它是一个可执行程序。

把脚本作为 ksh 的参数　如果没有把脚本设为可执行的，还可以把它作为参数传送给 ksh 命令，这样就可以执行该脚本。

范例 12-3

```
(命令行)
$ ksh myscript
```

说明

如果 ksh 命令带一个脚本名作为参数，它将执行该脚本。此时，脚本中的#!行将是不必要的，甚至不被用到。

一个脚本会话　在范例 12-4 中，用户将在编辑器中创建一个脚本。保存文件后，用户打开脚本的执行权限，然后执行它。如果程序中有任何错误，Korn shell 将立刻做出反应。

范例 12-4

(脚本)

```
1.  #!/bin/ksh
2   # This is the first Korn shell program of the day.
    # Scriptname: greetings
    # Written by: Karen Korny
3   print "Hello $LOGNAME, it's nice talking to you."
4   print "Your present working directory is $(pwd)."
    print "You are working on a machine called $(uname -n)."
    print "Here is a list of your files."

5   ls      # List files in the present working directory
    print "Bye for now $LOGNAME. The time is $(date +%T)!"
```

(命令行)

```
    $ chmod +x greetings
    $ greetings
3   Hello karen, it's nice talking to you.
4   Your present working directory is /home/lion/karen/junk
    You are working on a machine called lion.
    Here is a list of your files.
5   Afile          cplus     letter     prac
    Answerbook     cprog     library    prac1
    bourne         joke      notes      perl5
    Bye for now karen. The time is 18:05:07!
```

说明

1. 脚本的第一行 #!/bin/ksh 用于告诉内核将由哪个解释器来执行程序。
2. 以#号开头的注释不被执行。注释可以独立成行，也可以跟在命令的后面。
3. shell 完成变量替换后，print 命令将在屏幕上显示各行。
4. shell 完成命令替换后，print 命令将在屏幕上显示各行。
5. 执行 ls 命令。#号后的所有文本都是注释，将被 shell 忽略。

12.2 读取用户输入

read 命令用于从终端或文件读取输入，直至遇到一个新行。Korn shell 为 read 命令提供了一些额外的选项，表 12-1 列出了不同的读格式，表 12-2 则列出了读命令的选项。

表 12-1 read 命令格式

格　式	含　义
read answer	从标准输入读取一行并将其值赋给变量 answer
read first last	从标准输入读取一行，直至遇到第一个空白符或换行符。把用户键入的第一个词存到变量 first 中，而把该行的剩余部分保存到变量 last 中



(续表)

格　式	含　义
read response? "Do you feel okay?"	在标准错误输出显示"Do you feel okay?"，并等待用户输入回答，然后把回答赋值给变量 response。这种形式的 read 接收且只接收一个变量。无论用户键入什么内容，直至遇到新行，其内容都将被保存在 response 中
read –u3 line	读取文件描述符为 3 的文件，存入变量 line 中
read	读取输入，保存到内置的变量 REPLY 中

表 12-2　read 命令选项

选　项	含　义
-p	从协作进程读取输入的一行
– r	把换行符"\n"作为一个普通字符对待
– s	把一行复制到 history 文件中
– un	从文件描述 n 中读取，默认值为 fd 0 或标准输入
1988 版以后的 ksh 版中的选项	
– A	把字段存储为一个数组，数组下标从 0 开始
– t sec	对用户响应时间设置一个限制，单位为秒
– d char	用作从终端输入的一个可选分隔符，默认值为换行符

范例 12-5

(脚本)

```
    #!/bin/ksh
    # Scriptname: nosy
    print -n "Are you happy? "
1   read answer
    print "$answer is the right response."
    print -n "What is your full name? "
2   read first middle last
    print "Hello $first"
    print -n "Where do you work? "
3   read
4   print I guess $REPLY keeps you busy!
5   read place?"Where do you live? "
    # New ksh read and print combined
    print Welcome to $place, $first $last
```

(输出)

```
   $ nosy
    Are you happy? Yes
1   Yes is the right response.
2   What is your full name? Jon Jake Jones
    Hello Jon
3   Where do you work? Tandem
4   I guess Tandem keeps you busy!
5   Where do you live? Timbuktu
    Welcome to Timbuktu, Jon Jones
```

说明

1. read 命令接收一行用户输入，将其赋值给变量 answer。

2. read 命令从用户处接收输入，将输入的第一个词赋给变量 first，将第二个词赋给变量 middle，然后将直到行尾的所有剩余单词都赋给变量 last。

3. 不带参数的 read 命令，从用户处接收一行输入，并将输入赋值给内置变量 REPLY。

4. shell 完成变量替换后，print 函数将打印出字符串，显示出内置变量 REPLY 的值。

5. 如果 read 命令的参数末尾还附带一个问号(?)，则问号后的字符串将显示为一个提示。用户的输入将保存在变量 place 中。

12.2.1 read 命令和文件描述符

系统引导成功后，3 个称为流(stdin、stdout、stderr)的文件被打开，并赋值给一个文件描述符数组。最初的 3 个文件描述符 0、1、2 分别代表标准输入、标准输出和标准错误输出。接下来可用的文件描述符是描述符 3。选项 –u 允许 read 命令直接从文件描述符中读取内容。

范例 12-6

(命令行)

```
1    $ cat filex
     Captain Kidd
     Scarlett O'Hara
2    $ exec 3< filex       # filex is assigned to file descriptor 3 for reading
3    $ read -u3 name1      # read from filex and store input in variable, name1
4    $ print $name1
     Captain Kidd
5    $ read -u3 name2
     $ print $name2
     Scarlett O'Hara
6    $ exec 3<&-          # close file descriptor 3
7    $ read -u3 line
     ksh: read: bad file unit number
```

说明

1. 显示文件 filex 的内容。

2. 用 exec 命令打开文件描述符 3，从文件 filex 读取数据。

3. read 命令直接从单元 3(文件描述符 3，即 filex 的内容)读取一行，并将该行赋值给变量 name1。

4. 打印出变量 name1 中存储的值。

5. 文件 filex 仍然处于打开状态，read 命令从中读取下一行，并将该行赋值给变量 name2。

6. 关闭文件描述符 3(单元 3)，即关闭文件 filex。

7. read 命令试图从文件描述符 3 中读取输入内容，并赋值给变量 line，由于文件描述符 3(filex)已经关闭，读命令失败。



12.2.2 从整个文件中读取数据

范例 12-7 在一个 while 循环中使用 read 命令，循环将遍历整个文件，每次读取一行，直至读到文件末尾时循环结束。其中的文件由描述符(单元)打开用于读取数据。

范例 12-7

```
(文件)
1    $ cat names
     Merry Melody
     Nancy Drew
     Rex Allen
     $ cat addresses
     150 Piano Place
     5 Mystery Lane
     130 Cowboy Terrace
--------------------------------------------------------------------
(脚本)
     # !/bin/ksh
     # Scriptname: readit
2    while read -u3 line1 && read -u4 line2
     do
3       print "$line1:$line2"
4    done 3<$1 4<$2
--------------------------------------------------------------------
(命令行)
5    $ readit names addresses
     Merry Melody:150 Piano Place
     Nancy Drew:5 Mystery Lane
     Rex Allen:130 Cowboy Terrace
```

说明

1. 两个文件的内容，即所显示的姓名和地址。

2. while 循环开始。read 命令从文件描述符 3(单元 3)中读取一行输入，如果成功，再从文件描述符 4 中读取另一行输入。文件名作为参数或位置参量来传送。

3. 打印出变量值，格式为：第一个变量，冒号，第二个变量。

4. 给文件描述符 3 的输入，是第一个命令行参数 names；给文件描述符 4 的输入，是第二个命令行参数 address。

5. 执行带两个命令行参数(两个文件名)的脚本。

12.3 算术运算

Korn shell 支持整型运算和浮点运算，但浮点运算只在 1998 版后的版本中支持。typeset 命令用来给不同类型的变量赋值。表 12-3 是关于 typeset 命令的说明。

表 12-3 typeset 和运算

typeset 命令	别　名	含　义
type –i variable	integer variable	变量只能用整型赋值
typeset –i#		# 是整型数值的基数，如十进制、二进制等
1988 版以后的 ksh 版中的命令		
typeset –F variable		浮点数赋值
typeset –E variable	Float variable	浮点数赋值

12.3.1 整型数值

可以用 typeset –i 命令，或是它的别名 integer，来把一个变量声明为整型。如果试图给一个整型变量赋一个字符串值，ksh 将返回一个错误。如果是给它赋一个浮点值，则小数点及其后的小数将被截去。别名 integer 可用来代替 typeset –i。数字可以是基于不同进制的数，如二进制、八进制和十六进制。

范例 12-8

```
1   $ typeset -i num or integer num    # integer is an alias for typeset -i
2   $ num=hello
    /bin/ksh: hello: bad number
3   $ num=5 + 5
    /bin/ksh: +: not found
4   $ num=5+5
    $ echo $num
    10
5   $ num=4*6
    $ echo $num
    24
6   $ num="4 * 6"
    $ echo $num
    24
7   $ num=6.789
    $ echo $nu
    6
```

说明

1. 用带 - i 选项的 typeset 命令创建一个整型变量 num。
2. 试图把字符串 hello 赋值给整型变量 num，结果导致一个错误。
3. 在算术表达式中，如果没有使用(( ))操作符，空格必须用引号封起来或是删掉(参见 12.3.4 节“算术运算符和 let 命令”)。
4. 去掉空格后，运算正确执行。
5. 执行一个乘法运算，结果赋值给变量 num。
6. 空格包含在引号中，乘法运算可以执行，并且保证 shell 不会扩展通配符(*)。
7. 由于变量 num 被设置为整型，因此浮点数的小数部分被忽略。



12.3.2 使用不同的基数

typeset 命令使用 -i 选项，再带上数值的基数后，就可以用不同的进制来显示，包括十进制(基数为 10)、八进制(基数为 8)等，以此类推[①]。

范例 12-9

```
1   $ num=15
2   $ typeset -i2 num    # binary
    $ print $num
    2#1111
3   $ typeset -i8 num    # octal
    $ print $num
    8#17
4   $ typeset -i16 num   # hex
    $ print $num
    16#f
5   $ read number
    2#1101
    $ print $number
   2#1101
6   $ typeset -i number
    $ print $number
    2#1101
7   $ typeset -i10 number   # decimal
    $ print $number
    13
8   $ typeset -i8 number    # octal
    $ print $number
    8#15
```

说明

1. 变量 num 赋值为 15。
2. typeset 命令将数字转换为二进制格式。显示的是基数(2)后面跟一个#号，后面是数字的二进制值。
3. typeset 命令将数字转换为八进制格式。并显示数字的值。
4. typeset 命令将数字转换为十六进制格式。并显示数字的值。
5. read 命令从用户接收输入，用户输入的是二进制格式的数字，保存在变量 number 中，并以二进制显示。
6. typeset 命令把变量 number 转换为整数，仍以二进制显示。
7. typeset 命令把变量 number 转换为十进制整数，并显示其值。
8. typeset 命令把变量 number 转换为八进制整数，并显示为八进制数。

12.3.3 列出所有整型变量

typeset 命令带上参数 -i 将列出所有已设的整型变量及其值，如下所示。

---

① 在 1988 版以后的 Korn shell 版本中，基数可以大于 36。

```
$ typeset -i
ERRNO=2
LINENO=1
MAILCHECK=600
OPTIND=1
PPID=4881
RANDOM=25022
SECONDS=47366
TMOUT=0
n=5
number=#15
```

12.3.4 算术运算符和 let 命令

let 命令是 Korn shell 的一个内置命令，用来执行整型运算(参见表 12-4)。它代替了 Bourne shell 的整型测试。在使用 let 命令时，推荐使用其可选形式：(( ))操作符。

表 12-4 let 操作符[2]

<table>
<tr><th>操 作 符</th><th>含 义</th></tr>
<tr><td>-</td><td>负号</td></tr>
<tr><td>!</td><td>逻辑非</td></tr>
<tr><td>~</td><td>按位求反</td></tr>
<tr><td>*</td><td>乘法</td></tr>
<tr><td>/</td><td>除法</td></tr>
<tr><td>%</td><td>求模</td></tr>
<tr><td>+</td><td>加法</td></tr>
<tr><td>-</td><td>减法</td></tr>
<tr><td><<</td><td>左移位</td></tr>
<tr><td>>></td><td>右移位</td></tr>
<tr><td><= >= < > == !=</td><td>关系运算符</td></tr>
<tr><td>&</td><td>按位与</td></tr>
<tr><td>^|</td><td>异或</td></tr>
<tr><td>&&</td><td>逻辑与</td></tr>
<tr><td>||</td><td>逻辑或</td></tr>
<tr><td>!</td><td>逻辑非</td></tr>
<tr><td>=</td><td>赋值</td></tr>
<tr><td>*= /= %= += -= <<= >>= &= ^= |=</td><td>赋值简写符</td></tr>
</table>

② ++和--操作符在 1988 版以后的 ksh 版本中支持。



范例 12-10

```
1   $ i=5
2   $ let i=i+1
    $ print $i
    6
3   $ let "i = i + 2"
    $ print $i
    8
4   $ let "i+=1"
    $ print $i
    9
```

说明

1. 变量 i 赋值为 5。
2. let 命令使变量 i 的值增加 1。在执行运算时，不需要用美元符$来标识变量。
3. 参数带有空格时，就需要引号。
4. += 使变量 i 的值增加了 1。

范例 12-11

（命令行）

```
1   $ (( i = 9 ))
2   $ (( i = i * 6 ))
    $ print $i
    54
3   $ (( i > 0 && i <= 10 ))
4   $ print $?
    1
    $ j=100
5   $ (( i < j || i == 5 ))
6   $ print $?
    0
7   $ if (( i < j && i == 54 ))
    > then
    > print True
    > fi
    True
    $
```

说明

1. 变量 i 赋值为 9。操作符(())是 let 命令的可选形式，表达式括在双括号中，因此操作符、操作数之间允许有空格。

2. 变量 i 赋值为 i 与 6 的乘积。

3. 测试数值表达式。如果两个表达式值都为真，将返回状态 0。

4. ?是一个特殊变量，它的值是最后一个所执行命令(let 命令)的退出状态值，值为 1，因此该命令执行失败(等价于 false)。

5. 测试数值表达式。如果有一个表达式值为真，将返回退出状态 0。

6. ?是一个特殊变量，它的值是最后一个所执行命令(let 命令)的退出状态值，值为 0，因此该命令执行成功(等价于 true)。

7. 条件命令 if 后跟 let 命令，接着显示 shell 的次提示符，以等待命令执行完成。如果命令的退出状态为 0，则执行 then 语句后的命令。否则，将返回到 shell 的主提示符。

12.4 位置参量和命令行参数

用户可以在脚本中使用位置参量来引用命令行参数，例如，$1 代表第 1 个参数，$2 代表第 2 个参数，$3 代表第 3 个参数。位置参量可以用 set 命令重设。参见表 12-5。

表 12-5 位置参量

变 量	功 能
$0	指代脚本名
$#	代表位置参量的个数
$*	是一个包含所有位置参量的列表
$@	未加双引号时，与$*的含义相同
"$*"	扩展为单个变量，例如："$1 $2 $3"
"$@"	扩展为多个单独的变量，例如："$1"、"$2"、"$3"

set 命令与位置参量

可以用 set 命令来设置位置参量。如果是已经被设置的位置参量，set 命令将重新设置它，并清除位置参量列表中原有的值。可使用命令 set --来清除所有的位置参量。

范例 12-12

(脚本)

```
    $ cat args
    # !/bin/ksh
    # Script to test command-line arguments
1   print The name of this script is $0.
2   print The arguments are $*.
3   print The first argument is $1.
4   print The second argument is $2.
5   print The number of arguments is $#.
6   oldparameters= $*
7   set Jake Nicky Scott
8   print All the positional parameters are $*.
9   print The number of positional parameters is $#.
10  print $oldparameters
11  set --
12  print Good-bye for now, $1.
13  set $oldparameters
```



```
14  print $*
    (输出)
    $ args a b c d
1   The name of this script is args.
2   The arguments are a b c d.
3   The first argument is a.
4   The second argument is b.
5   The number of arguments is 4.
8   All the positional parameters are Jake Nicky Scott.
9   The number of positional parameters is 3.
10  a b c d
12  Good-bye for now ,.
14  a b c d
    $
```

说明

1. 脚本的名字保存在变量$0 中。
2. $*(和$@)代表所有的位置参量。
3. $1 代表第 1 个位置参量(命令行参数)。
4. $2 代表第 2 个位置参量。
5. $#是位置参量(命令行参数)的总个数。

6. 把所有的位置参量($*)都保存在变量 oldparameters 中，在后面的操作中，如果想取回初始参量值，使用命令 set $oldparameters 就可以了。

7. set 命令重置位置参量，清空原来的位置参量列表。$1 赋值为 Jake，$2 赋值为 Nicky，$3 则赋值为 Scott。

8. 打印新的位置参量。
9. 打印位置参量的个数。
10. 初始位置参量保存在变量 oldparameters 中，打印其值。
11. 清空所有参量。
12. 参量$1 没有值，因为参量列表已被命令 set -- 清空。
13. 用 set 命令为新的参量列表赋值，用变量 oldparameters 中的值替换参量列表。
14. 打印所有的位置参量。

范例 12-13

```
($*和$@的区别)
1   $ set 'apple pie' pears peaches
2   $ for i in $*
    > do
    > echo $i
    > done
    apple
    pie
    pears
    peaches
3   $ set 'apple pie' pears peaches
```

```
4   $ for i in "$*"
    > do
    > echo $i
    > done
    apple pie pears peaches

5   $ set 'apple pie' pears peaches
6   $ for i in $@
    > do
    > echo $i
    > done
    apple
    pie
    pears
    peaches

7   $ set 'apple pie' pears peaches
8   $ for i in "$@"          # At last!!
    > do
    > echo $i
    > done
    apple pie
    pears
    peaches
```

说明

1. 设置命令参量。扩展 $* 时，将去掉引号，apple 和 pie 是两个单独的词。for 循环依次将每个词赋值给变量 i，然后打印变量 i 的值。每次循环中，左边的词就被移掉，下一个词被赋值给 i。
2. $* 由双引号括起来，则其中的所有词组成一个字符串，然后赋值给变量 i。
3. 设置位置参量。
4. 把 $* 括在双括号中，则整个位置参量列表组成一个字符串。
5. 设置位置参量。
6. 不带引号，$@ 与 $* 用法一样。
7. 设置位置参量。
8. $@ 用双引号括起来，则每个位置参量分别被当成一个带双引号的字符串，列表将由“apple pie”、“pears”和“peaches”组成。在循环的每次递归中，每个带双引号的词依次赋值给 i。

12.5 分支结构和流程控制

使用分支指令可以基于所给条件的成功与否来执行相应的任务。if 命令是最简单的判定方式。if/else 命令是一种二路判定结构，if/elif/else 命令提供了多路判定结构。

Korn shell 希望 if 后跟一个命令。该命令可以是一个系统命令或内置命令，命令的退



出状态将用于条件判断。计算表达式可用内部命令 test，该命令也被链接到符号 [和 [[。Bourne shell 用单括号 [和] 来封装一个表达式，而 Korn shell 则用一种更高级的方法来测试表达式，表达式用双括号 [[和]] 来封装。在单括号中，不允许通配符的扩展，而双括号(只在 Korn shell 中)不但支持通配符的扩展，同时还增加了一套新的操作符。测试命令的结果是一个整数值，状态 0 表示成功，非 0 表示失败。

12.5.1 测试退出状态和$?变量

变量?的值是一个 0~255 之间的整数，表示最后一条命令的退出状态。如果退出状态值为 0，则命令是成功退出，如果非 0，则命令因为某种原因导致失败。可以测试命令的退出状态，也能用 test 命令来检查表达式的退出状态。

下面的范例将演示如何对退出状态进行检测。在 Bourne shell 中使用的是单括号，在 Korn shell 中也完全可以使用，同时，Korn shell 中还可以用双括号来检测表达式。

范例 12-14

(命令行)

```
1   $ name=Tom
2   $ grep "$name" datafile
    Tom Savage:408-124-2345
3   $ print $?
    0                             # Success
4   $ test $name = Tom
5   $ print $?
    0                             # Success
6   $ test $name != Tom
    $ print $?
    1                             # Failure
7   $ [ $name = Tom ]             # Brackets instead of the test command
8   $ print $?
    0
9   $ [[ $name = [Tt]?m ]]        # New ksh test command
10  $ print $?
    0
```

说明

1. 将字符串 Tom 赋给变量 name。
2. grep 命令在 datafile 文件中搜索字符串 Tom，如果找到，就显示找到的行。
3. 变量?，通过$? 方式来访问，包含了最后被执行的命令的退出状态，在这种情况下是指 grep 的退出状态。如果 grep 命令成功搜索到字符串 Tom，则返回 0 状态。表明 grep 命令成功。
4. test 命令用于测试字符串和数字，也用于文件的测试。如果表达式为真则返回 0 状态，否则返回 1。等号前后必须有空格。
5. 测试 name 的值是否等于 Tom。test 命令返回 0 状态，意味着$name 的值等于 tom。
6. 测试 name 的值是否等于 Tom。test 命令返回 1 状态，意味着$name 的值不等于 tom。
7. 方括号是 test 命令的替代符号。第一个方括号后面必须有空格。表达式用于测试

$name 的值是否等于 Tom。

8. test 的退出状态为 0。$name 的值等于 Tom，因此 test 命令成功。

9. 使用 Korn shell 新增的测试命令 [[。新测试命令 [[允许 shell 进行元字符扩展。如果变量与形如 tom，Tom，tim，Tim 的字符串相匹配，则返回成功状态 0。

10. 变量 name 匹配以 T 或 t 开头、以 m 结尾的字符串，返回成功状态 0(返回状态保存在变量 $? 中)。

12.5.2 老的 test 命令

test 命令用于测试条件表达式并返回‘真’或‘假’。返回退出状态 0 表示‘真’，退出状态非 0 则表示‘假’。test 命令或方括号都可以用。Korn shell 增加了一种新的测试表达式的方法，使用双方括号。为了和 Bourne shell 兼容，较老形式的测试则是使用 test 命令或单方括号。但是，Korn shell 推荐使用双方括号式的新 test 命令。test 操作符(包括老模式和新模式)的完整列表参见表 12-6。

表 12-6 测试和逻辑操作符

测试格式	测试内容
字符串测试：	
string1 = string2	string1 等于 string2
string1 != string2	string1 不等于 string2
string	string 不为空
- z string	string 的长度为 0
- n string	string 的长度为非 0
例如：test –n $word 或 [- n $word] test tom = suc 或 [tom = suc]	
整数测试(在 Bourne shell 中使用的老式风格的测试)：	
int1 –eq int2	int1 等于 int2
int1 –ne int2	int1 不等于 int2
int1 –gt int2	int1 大于 int2
int1 –ge int2	int1 大于或等于 int2
int1 –lt int2	int1 小于 int2
int1 –le int2	int1 小于或等于 int2
逻辑操作符(老式风格的测试)：	
!	‘非’操作符
- a	‘与’操作符
- o	‘或’操作符
文件测试(老式风格的测试)：	
- b filename	块专用文件
- c filename	字符专用文件

(续表)

测 试 格 式	测 试 内 容
- d filename	目录存在
- f filename	文件存在且不是目录
- g filename	Set - group - ID 被设置
- h filename	符号链接
- k filename	Sticky 位被设置
- p filename	文件是一个命名管道
- r filename	文件可读
- s filename	文件大小非 0
- u filename	Set - user - ID 位被设置
- w filename	文件可写
- x filename	文件可执行

12.5.3 新的 test 命令

使用方括号的 test 命令，可以使用一些新增的操作符。还可以在字符串匹配测试中使用通配符，而且也更正了老 test 命令中的许多错误。新的字符串测试操作符见表 12-7。

表 12-7 字符串测试(新模式测试)

字符串测试操作符	测 试 内 容
string = pattern	string 匹配 pattern[③]
string != pattern	string 不匹配 pattern
string1 < string2	string1 的 ASCII 值小于 string2
string1 > string2	string1 的 ASCII 值大于 string2
- n string	string 的长度非 0，有参数
- z string	string 的长度为 0，无参数

范例 12-15

```
(脚本)
    read answer
1   if [[ $answer = [Yy]* ]]     # Test for Yes or yes or Y or y, etc.
    then...
    Example:
    (脚本)
    guess=Noone
2   if [[ $guess != [Nn]o@(one|body) ]]
                               # Test for Noone, noone, or Nobody, nobody...
    then. . .
    Example:
```

③ 1988 年以后的版本，操作符 '==' 也允许出现。

```
    (命令行)
3   [[ apples < oranges ]]
    print $?
    0
4   [[ apples > oranges ]]
    print $?
    1
5   $ name="Joe Shmoe"
    $ [ $name = "Abe Lincoln" ]           # old style
    ksh: Shmoe: unknown test operator
6   $ [[ $name = "Abe Lincoln" ]]         # new style
    $ echo $?
    1
```

说明

1. 测试用户输入的变量 answer，看它是否是以 Y 或 y 为开头的字符串。

2. 测试变量 guess。如果它不是一个以 N 或 n 为开头，后跟 o，最后以 one 或 body 结尾的字符串(比如，noone 或 noboay)，则执行 then 后面的命令。

3. 测试字符串 apples 在 ASCII 排序表中是否排在 oranges 之前。结果为是。

4. 测试字符串 apples 在 ASCII 排序表中是否排在 oranges 之后。结果为否。

5. 在老式风格的 test 中，变量 name 被拆分为多个独立的单词。因为 '=' 操作符只允许单个字符串作为其左操作数，故 test 命令失败。为了解决这种问题，变量应该用双引号括起来。

6. 在新式风格的 test 中，变量不用拆分为多个独立单词。因此$name 不需要再用双引号括起来。

12.5.4 带有二元操作符的文件测试

测试文件的二元操作符需要有两个操作数(也就是说，操作符的两边各有一个文件)。二元文件测试操作符列表见表 12-8。

表 12-8 二元文件测试和逻辑操作符

操 作 符	测 试 内 容
file1 –nt file2	如果 file1 比 file2 新则为真
file1 –ot file2	如果 file1 比 file2 老则为真
file1 –ef file2	如果 file1 是 file2 的另一个名字则为真

12.5.5 逻辑操作符

Korn shell，就像 C shell 一样，提供对表达式'真'或'假'的逻辑测试。见表 12-9。



表 12-9　逻辑操作符

操　作　符	测 试 内 容
&&	与操作符。测试 && 左边的表达式，若为真，则测试&&右边的表达式，并且必须也为真，表达式才为真。如果有一个表达式为假，则表达式即为假。 && 操作符取代了 -a 选项，例如：((($x && $y)>5))
||	或操作符。测试 || 左边的表达式，若为真。则表达式为真；若为假，则测试 || 右边的表达式，若为真，则表达式为真。只有当两个表达式都为假时表达式才为假。 || 操作符取代了 -o 选项，例如：((($x || $y)))

12.5.6　文件测试

为检测文件的属性，Korn shell 提供了一系列内置测试命令，例如：测试文件是否存在，测试文件的类型、权限等。文件测试选项(也称为标志)如表 12-10 所示。

表 12-10　文件测试(新的 test 标志)

测 试 标 志	测 试 内 容
只用于 Korn Shell：	
-a file	file 存在
-e file	file 存在(1988 年以后的版本)
-L file	file 存在并且是一个符号链接
-O file	您是 file 的所有者
-G file	您的 group ID 和 file 的相同
-S file	file 存在并且是一个 socket
用于 Bourne 和 Korn Shell：	
-r file	file 存在且可读
-w file	file 存在且可写
-x file	file 存在且可执行
-f file	file 存在且不是目录
-d file	file 存在且是一个目录
-b file	file 存在且是一个块专用文件
-c file	file 存在且是一个字符专用文件
-p file	file 存在且是一个命名的管道
-u file	file 存在且被设置了 uid
-g file	file 存在且被设置了 gid
-k file	file 存在且 sticky 位被设置了
-s file	file 长度非 0

范例 12-16

(脚本)

```
1 file=/etc/passwd
2 if [[ -f $file && (-r $file || -w $file) ]]
  then
3   print $file is a plain file and is either readable or writable
  fi
```

说明

1. 变量 file 被赋值为/etc/passwd。

2. 文件测试操作符测试文件是否是一个普通文件并且是可读或可写的。圆括号用于分组。在老的 test 中，圆括号必须用反斜杠进行转义。

3. 如果两个测试均为真，则文件是一个普通文件，并且是可读或可写的，于是该行被执行。

12.5.7 if 命令

条件的最简单形式就是 if 命令。跟在关键字 if 后面的命令被执行，并返回它的退出状态。如果退出状态为 0 则该命令成功，并且关键字 then 后面的语句被执行。

在 C shell 和 C 语言中，跟在 if 命令后的表达式是一个布尔类型的表达式，但在 Bourne shell 和 Korn shell 中，跟在 if 后面的语句是一条或一组命令，if 行最后一个命令的退出状态用于决定是否要继续执行 then 语句后面的命令。如果 if 行最后一个命令的退出状态为 0，则 then 后面的命令被执行，fi 使得 then 后面的命令停止执行。如果退出状态非 0，说明命令失败，则 then 后面的语句被忽略并且控制会直接转到 fi 语句之后。

条件命令可以嵌套。每一个 if 必须有一个相应的 fi，fi 和最近的 if 配对。采用缩进方式来格式化显示 if 块将有助于程序的调试。

格式

```
if command
then            # testing command exit status
    command
    command
fi
--------------------------------------------

if test expression
then        # Using the test command to test expressions
   command
fi
```

或

```
if [ expression ]
then            # Using the old-style test command--
   command      # brackets replace the word test
fi
--------------------------------------------

if [[ expression ]]
then            # New-style brackets for testing expressions
```



```
    command
fi
---------------------------------------------
 if command
 then
 ...
  if command
  then
  ...
   if command   # Nested conditionals
   then
   ...
   fi
  fi
fi
```

范例 12-17

```
1   if ypmatch $name passwd > /dev/null 2>&1
2   then
        echo Found $name!
3   fi
```

说明

1. ypmatch 是一个 NIS 命令，用来搜索命令的参数 name，该参数位于服务器上的 NIS passwd 数据库中。标准输出和标准错误输出被重定向到 UNIX 的位容器/dev/null。
2. 如果 ypmatch 命令的退出状态为 0，程序转到 then 语句并执行命令直到到达 fi。
3. fi 标志着 then 语句后面的命令结束。

12.5.8 使用老式风格的 Bourne test

如果您一直使用 Bourne shell 编程，由于 Korn shell 是向后兼容的，所以 Bourne shell 脚本可以被 Korn shell 正常执行。转换到 Korn shell 之后，很多 Bourne shell 程序员在测试表达式时仍然使用老模式的 test 命令。如果您在阅读或维护脚本，您会发现老的语法依然适用并且没有任何问题。所以，简单地讨论一下老的语法是有必要的，即使您正在用新的 Korn shell test 命令来编写脚本。

范例 12-18

```
    # !/bin/ksh
    # Scriptname: are_you_ok
1   print "Are you ok (y/n) ?"
    read answer
2   if [ "$answer" = Y -o "$answer" = y ]     # Old-style test
    then
       print "Glad to hear it."
3   fi
```

说明

1. 用户被询问 'Are you ok (y/n)?'。read 命令使得程序停下来等待用户输入。

2. test 命令，由一个 [表示，用来测试表达式并返回退出状态，表达式为真返回 0，为假则返回非 0。如果变量 answer 测试为 Y 或 y，则 then 语句后面的命令被执行(老式风格下进行表达式测试时，test 命令不允许使用通配符)。

3. fi 标志着 then 语句后面的命令结束。

12.5.9 使用新式风格的 Korn test

新式风格的 Korn shell test 命令允许表达式包含 shell 元字符和 Korn shell 操作符，例如 &&、||。

范例 12-19

```
   #!/bin/ksh
   # Scriptname: are_you_ok2
1  print "Are you ok (y/n) ?"
   read answer
2  if [[ "$answer" = [Yy]* ]]        # New-style test
   then
       print "Glad to hear it."
3  fi
```

说明

1. 用户被询问‘Are you ok (y/n)?’。read 命令使得程序停下来等待用户的输入。

2. [[ ]] 是一个用于测试表达式的特殊的 Korn shell 结构。如果变量 answer 的值为一个以 Y 或 y 为开头的字符串，then 语句后面的命令被执行。

3. fi 终止 if 语句。

12.5.10 使用旧式风格的带数字表达式的 Bourne test

为了测试带数字的表达式，Korn shell 仍然可以使用旧式的 Bourne shell 的 test 命令和操作符，但是最好使用新式的 let 命令。

范例 12-20

```
1   if [ $# -lt 1 ]
    then
       print "$0: Insufficient arguments " 1>&2
       exit 1
2   fi
```

说明

1. 该语句含义为：如果参数的个数小于 1，则打印错误信息并将它发送到标准输出，然后退出程序。这是用老式风格的 test 命令来测试整数。

2. fi 标志着 then 之后的语句块的结束。

12.5.11 let 命令和数字测试

现在仍然可以使用单方括号和老式风格的 Bourne shell 数字操作符，来测试数字表达



式，但 Korn shell 推荐使用双圆括号和新的 C 语言风格的数字操作符来测试数字表达式。要注意的是：双方括号只用于字符串表达式测试和文件测试(参见表 12-10)。

范例 12-21

```
1    if (( $# < 1 ))
     then
        print "$0: Insufficient arguments " 1>&2
        exit 1
2   fi
```

说明

1. 该语句含义为：如果参数的个数小于 1，则显示错误信息并将它发送到标准输出，然后退出程序。在 Korn shell 中，这是执行数字测试时推荐使用的方法。

2. fi 标志着 then 后面的语句块结束。

12.5.12 if/else 命令

if/else 命令将允许一个二路分支处理。如果 if 后面的命令失败，则 else 后的命令被执行。

格式

```
if command
then
    command (s)
else
    command (s)
fi
```

范例 12-22

```
1   if ypmatch "$name" passwd > /dev/null 2>&1
2   then
        print Found $name!
3   else
4       print "Can't find $name."
        exit 1
5   fi
```

说明

1. ypmatch 在 NIS passwd 数据库中搜索命令的参数$name。标准输出和错误被重定向到 UNIX 位容器/dev/null。

2. 如果 ypmatch 命令的退出状态为 0，则程序控制转到 then 语句并执行命令，直至 else。

3. 如果 ypmatch 命令在 passwd 数据库中没有找到$name，则执行 else 语句后面的命令。即 ypmatch 命令的退出状态必须非 0，else 后的命令才会被执行。

4. print 函数把输出发送到屏幕，程序退出。

5. fi 标志着 if 结构的结束。

12.5.13 if/elif/else 命令

if/elif/else 命令允许多路分支处理。如果 if 后的命令失败，则测试 elif 后的命令；如果 elif 后的命令成功，则执行与它对应的 then 后的命令。如果 elif 后的命令也失败，测试下一个 elif 后的命令。如果没有命令成功，则执行 else 后的命令。else 块为默认部分。

格式

```
if command
then
    command(s)
elif command
then
    command(s)
elif command
then
    command(s)
else
    command(s)
fi

if [[ string expression ]] or   if (( numeric expression ))
then
    command(s)
elif [[ string expression ]]or   elif (( numeric expression ))
then
    command(s)
elif [[ string expression ]]or   elif (( numeric expression ))
then
    command(s)
else
    command(s)
fi
```

范例 12-23

(脚本)

```
  #!/bin/ksh
  # Scriptname: tellme
1  read age?"How old are you? "
2  if (( age < 0 || age > 120 ))
   then
       print "Welcome to our planet! "
       exit 1
   fi
3  if (( age >= 0 && age < 13 ))
   then
       print "A child is a garden of verses"
   elif (( age > 12 && age < 20 ))
   then
       print "Rebel without a cause"
```




```
   elif (( age >= 20 && age < 30 ))
   then
       print "You got the world by the tail!!"
   elif (( age >= 30 && age < 40 ))
   then
       print "Thirty something..."
4  else
       print "Sorry I asked"
5  fi
```

(输出)

```
   $ tellme
1  How old are you? 200
2  Welcome to our planet!
   $ tellme
1  How old are you? 13
3  Rebel without a cause
   $ tellme
1  How old are you? 55
4  Sorry I asked
```

说明

1. 请求用户输入。输入的值赋给变量 age。

2. 在双圆括号中执行数字测试。如果 age 小于 0 或大于 120，执行 print 命令并终止程序的运行返回退出状态值 1，然后回到 shell 提示符。注意，当使用双括号操作符时，不需要使用 $ 符号来作变量替换。

3. 在双圆括号中执行数字测试。如果 age 大于 0 并小于 13，let 命令返回退出状态 0，即为真。

4. else 结构是默认部分。如果上面的语句都为假，则执行 else 后面的命令。

5. fi 终止最外层的 if 语句。

12.5.14 exit 命令

exit 命令用于终止程序并返回命令行。您可以在有些条件不为真时，使脚本退出。exit 命令的参数是一个整数，范围为 0~255。当程序退出时，退出值保存在 shell 的 ? 变量中。

范例 12-24

(脚本)

```
   #!/bin/ksh
   # Scriptname: filecheck
   # Purpose: Check to see if a file exists, what type it is, and its
               # permissions.
1  file=$1     # Variable is set to first command-line argument
2  if [[ ! -a $file ]]
   then
       print "$file does not exist"
       exit 1
   fi
3  if [[ -d $file ]]
```

```
    then
        print "$file is a directory"
4   elif [[ -f $file ]]
    then
5      if [[ -r $file && -w $file && -x $file ]]
       then
          print "You have read, write, and execute permission on
          file $file"
       else
6         print "You don't have the correct permissions"
          exit 2
       fi
   else
7      print "$file is neither a file nor a directory. "
       exit 3
8   fi
(命令行)
9   $ filecheck testing
    testing does not exist
10  $ echo $?
    1
```

说明

1. 传给程序的第一个命令行参数($1)被赋给变量 file。

2. 跟在 if 后的 test 命令对文件进行测试，如果 $file(在变量替代之后)是一个不存在的文件(注意取反操作符“!”)，则执行关键字 then 后面的命令。退出值为 1 意味着程序失败(在这种情况下，测试失败)。

3. 如果 file 是一个目录，则打印它是目录。

4. 如果 file 不是目录，且文件是一个普通文件，则继续向下执行。

5. 如果文件是可读、可写和可执行的，则继续向下执行。

6. fi 终止最内层的 if 命令。如果文件不具有读、写和执行的权限，则程序以 2 为参数退出。

7. 如果第 2 行和第 3 行失败，则执行 else 命令。程序带以 3 为参数退出。

8. fi 和例子中第 3 行的 if 相对应。

9. 名为 testing 的文件不存在。

10. 变量 $? 中保存着退出状态值 1。

12.5.15 null 命令

null 命令是一个冒号。它是一个内置的、什么都不做的命令，返回退出状态 0。当你不想执行任何操作，但又必须有一条命令，或是 then 语句后必须跟一些语句否则程序将产生一个错误信息时，它将作为一个占位符用在 if 命令之后。null 命令经常用作 loop 命令的一个参数，使循环成为一个死循环，或用来测试变量表达式的修饰符，例如 {EDITOR: -/bin/vi}。



范例 12-25

(脚本)

```
1   name=Tom
2   if grep "$name" databasefile > /dev/null 2>&1
    then
3       :
4   else
        print "$1 not found in databasefile"
        exit 1
    fi
```

说明

1. 字符串 Tom 赋给变量 name。

2. if 命令测试 grep 命令的退出状态。如果在 databasefile 中找到 Tom，则执行 null 命令，即不做任何操作。

3. 冒号即 null 命令，null 命令始终以 0 退出状态退出。

4. 如果没有找到 Tom，就打印错误信息并退出。如果 grep 命令失败，else 后的命令将会被执行。

范例 12-26

(脚本)

```
1    : ${EDITOR:=/bin/vi}
2    echo $EDITOR
```

说明

1. 冒号带有一个由 shell 计算出来的参数。表达式 {EDITOR:=/bin/vi} 作为冒号命令的一个参数来使用。如果变量 EDITOR 在前面已经被设置了，则它的值不会被改变。如果还没被设置过，/bin/vi 值将会赋给它。如果冒号不在表达式之前，Korn shell 将会报告错误，如 ksh：/bin/vi not found。

2. 显示变量 EDITOR 的值。

12.5.16 case 命令

case 命令是一个多路分支命令，可用来代替 if/elif 命令。case 变量的值和 value1、value2 等进行比较，直到找到一个匹配的值为止。当某一个值匹配 case 变量时，跟在该值后面的命令会被执行，直到到达双分号为止。然后，指令从 esac(case 的反向拼写)之后开始继续执行。

如果 case 变量找不到匹配，程序执行“*)”之后的命令，即默认的命令，直到到达双分号或 esac。“*)”值和 if/else 条件中的 else 语句起着同样的作用。case 值可以使用 shell 通配符，也可以使用竖线“|”(管道符)来对两个值进行或运算。

格式

```
case variable in
value1)
     command(s);;
```

```
value2)
    command(s);;
*)
    command(s);;
esac
```

范例 12-27

(脚本)

```
    #!/bin/ksh
    # Scriptname: xtermcolor
    # Sets the xterm foreground color (the color of the prompt and
    # input typed for interactive windows.
1   read color?"Choose a foreground color for your terminal?"
2   case "$color" in
3   *[Bb]l??)
4      xterm -fg blue -fn terminal &
5          ;;
6   *[Gg]reen)
        xterm -fg darkgreen -fn terminal &
          ;;
7   red | orange)               # The vertical bar means "OR"
       xterm -fg "$color" -fn terminal &
          ;;
8   *)  xterm -fn terminal &    # default
          ;;
9   esac
10  print "Out of case..."
```

说明

1. 请求用户输入。用户的输入赋给变量 color。
2. 用 case 命令测试变量 $color 的值。
3. 如果变量 color 以 B 或 b 开头，后跟字符 l 和任意两个字符，case 表达式匹配第一个值。该值以一个简单的右圆括号结束。这些通配符是 shell 元字符。
4. 如果第 3 行的值匹配 case 表达式，则执行该语句。xtem 命令把前景颜色设置为蓝色。
5. 在命令块的最后一条命令之后必须有一个双分号。到达该分号后，控制跳转到第 10 行。
6. 如果 case 表达式匹配为以 G 或 g 开头、后跟字符串 reen，则 xtem 窗口的前景颜色设置为深绿色。双分号终止语句块，并将控制分支跳转到第 10 行。
7. 符号“|”作为或操作符使用。如果 case 表达式和 red 或 orange 相匹配，执行 xtem 命令。
8. 这是默认值。如果上面的值都不能和 case 表达式相匹配，则执行*) 之后的命令。终端前景色的默认值为黑色。
9. esac 语句终止 case 命令。
10. 完成前面的值匹配后，继续从这里执行。

case 命令和 here 文档　通常，here 文档用于生成一个菜单，当用户从菜单中做出选择之后，case 命令用来与之匹配。Korn shell 也为生成菜单提供一个 select 循环。

范例 12-28

```
(.profile 文件 )
    print "Select a terminal type "
1   cat << EOF
        1) vt120
        2) wyse50
        3) ansi
        4) sun
2   EOF
3   read TERM
4   case "$TERM" in
    1)  export TERM=vt120
        ;;
    2)  export TERM=wyse50
        ;;
    3)  export TERM=ansi
        ;;
    *)  export TERM=sun
        ;;
5   esac
    print "TERM is $TERM"
```

说明

1. here 文档用来显示一个选择菜单。
2. EOF 是用户定义的一个终止符。标志着 here 文档的输入到这里结束。
3. read 命令等待用户的输入，并把该输入赋给变量 TERM。
4. case 命令测试变量 TERM，并将它和列表中的数字进行匹配。如果找到匹配，则设置该终端。
5. case 命令由 esac 终止。

12.6 循环命令

循环命令用于多次执行一条命令或一组命令，一直执行，直到满足某个条件。Korn shell 有 4 种类型的循环：for 循环、while 循环、until 循环以及 select 循环。

12.6.1 for 命令

for 循环命令，用于为参数表中的每一个参数执行一组命令。您可以在一个文件列表或用户名列表上用该循环来执行一组相同命令。for 命令后，紧跟一个由用户定义的变量，接着是关键字 in，最后是一个字列表。第一次循环，字列表中的第一个字被赋给变量，然后移出第一个字，这样列表中的字就减少一个。下一次循环把第二个字赋给变量，然后把它

移出，以此类推。循环体从关键字 do 开始，到关键字 done 结束。当列表中的所有字都被移出时，循环结束，然后程序控制继续执行关键字 done 之后的命令。

格式

```
for variable in wordlist
do
  命令
done
```

范例 12-29

```
(脚本)
1    for pal in Tom Dick Harry Joe
2    do
3        print "Hi $pal"
4    done
5    print "Out of loop"
(输出)
     Hi Tom
     Hi Dick
     Hi Harry
     Hi Joe
     Out of loop
```

说明

1. for 循环将重复从名字列表中读取数据：Tom、Dick、Harry 和 Joe，依次将它们赋给变量 pal 同时将元素左移。当所有的名字都被移出且名字列表为空时，循环结束，从 done 关键字后继续执行。在循环中，pal 的值是变化的，第一次循环，pal 变量被赋值为 Tom，第二次循环被赋值为 Dick，最后一次被赋值为 Joe。

2. 名字列表之后必须有关键字 do。如果它和名字列表处于同一行，列表语句必须以分号结束。例如：for pal in Tom Dick Harry Joe; do。

3. 这是循环体。pal 变量被赋值后，循环体中的命令，即关键字 do 和 done 之间的所有命令将被执行。

4. 关键字 done 终止循环体，如果第 1 行的词列表中没有词可处理，则退出循环，从第 5 行开始执行。

5. 循环结束时，这一行被执行。

范例 12-30

```
(命令行)
1    $ cat mylist
     tom
     patty
     ann
     jake
(脚本)
2    for person in $(< mylist)     # same as for person in 'cat mylist'
     do
```



```
3       mail $person < letter
        print $person was sent a letter.
4   done
5   print "The letter has been sent."
```

说明

1. 显示 mylist 文件的内容。

2. 执行命令替换，mylist 文件的内容展开为词表。第一次循环，将 tom 赋值给变量 person，然后把 tom 从词表中移出。第二次循环则赋值为 patty，以此类推。

3. 在循环体中，向每个用户发送一个邮件：文件 letter 的副本。

4. done 关键字标志着循环体结束。

5. 向列表中的所有用户都已发送邮件后，循环结束，执行此行。

范例 12-31

```
1   for file in *.c
2   do
       if [[ -f $file ]] ; then
          cc $file -o ${file%.c}
       fi
    done
```

说明

1. 词表由当前工作目录下的所有扩展名为 .c 的文件(C 源文件)组成。在循环的每次遍历中，各文件名将被依次赋给变量 file。

2. 进入循环体时，将会测试文件是否存在。如果是存在，文件将会被编译，${file%.c} 将被扩展为不带.c 后缀的文件名。

12.6.2 词表中的变量$*和 $@

扩展以后，在不加双引号时 $* 和 $@ 含义相同，带双引号后，“$*”表示一个字符串，“$@”表示一个由独立单词组成的列表。

范例 12-32

```
(脚本)
    #!/bin/ksh
1   for name in $*      # or for name in $@
2   do
        echo Hi $name
3   done
(命令行)
$ greet Dee Bert Lizzy Tommy
Hi Dee
Hi Bert
Hi Lizzy
Hi Tommy
```

说明

1. $*和$@ 被扩展为位置参量表，在本例中，参数从命令行传递：Dee、Bert、Lizzy 和 Tommy，列表中的名字在 for 循环中依次赋给变量 name。

2. 执行循环体中命令，直到参数列表为空。

3. done 关键字标志着循环体结束。

12.6.3 while 命令

while 命令检查紧随其后的命令的值，如果命令的退出状态为 0，循环体内的命令(do 和 done 之间的命令)被执行。当遇到关键字 done 时，控制返回到循环开始处。while 命令再次检查命令的退出状态，循环一直重复，直到 while 命令检查到退出状态不为 0 时，循环结束，程序从 done 关键字后面的语句开始执行。如果命令退出状态一直为零，则循环将一直执行(当然，可以用 Ctrl+c 或者 Ctrl+\组合键来终止循环)。

格式

```
while command
do
   command(s)
done
```

范例 12-33

(脚本)

```
1   num=0                      # Initialize num
2   while (( num < 10 ))       # Test num with the let
    do
      print -n $num
3     (( num=num + 1 ))        # Increment num
    done
    print "\nAfter loop exits, continue running here"
```

(输出)

```
0123456789
After loop exits, continue runging here
```

说明

1. 初始化，变量 num 被赋值为 0。

2. while 命令后是 let 命令。如果变量 num 的值小于 10，就进入循环体。

3. 在循环体中，num 的值加 1。如果 num 的值保持不变，循环将无休止地递归，直至进程被终止。

范例 12-34

(脚本)

```
    #!/bin/ksh
    # Scriptname: quiz
1   read answer?"Who was the U.S. President in 1992? "
2   while [[ $answer != "Bush" ]]
3   do
```




```
          print "Wrong try again!"
4         read answer
5     done
6     print Good guess!
```

(输出)

```
$ quiz
Who was the U.S. President in 1992? George
Wrong try again!
Who was the U.S. President in 1992? I give up
Wrong try again!
Who was the U.S. President in 1992? Bush
Good guess!
```

说明

1. read 命令打印问号(?)后的字符串“Who was the U.S. President in 1992?”，然后等待用户输入，并将输入存入变量 answer 中。

2. 进入 while 循环，用测试命令 [[检测表达式的值。如果变量 answer 的值不是 Bush，就进入循环体，执行 do 和 done 之间的命令。

3. 关键字 do 是循环体的开始。

4. 要求用户重新输入。

5. 关键字 done 标志着循环体结束。控制转向 while 循环开始处，并再次检测表达式的值，如果$answer 的值不是 Bush，循环继续执行。当用户输入为 Bush 时，循环结束，程序控制跳转到第 6 行。

范例 12-35

(脚本)

```
1    go=1
     print Type q to quit.
2    while let go or (( go ))
     do
        print I love you.
        read word
3       if [[ $word = [qQ]* ]]
        then
           print "I'll always love you"
4          go=0
        fi
5    done
```

(输出)

```
$ sayit
Type q to quit.
I love you.
I love you.
I love you.
I love you.
I love you.
q
```

```
I'll always love you
$
```

说明

1. 变量 go 被赋值为 1。

2. 进入循环后，test 命令检测表达式的值，表达式值为 1，程序进入 while 循环。

3. 如果用户输入 q 和 Q 给变量 word，则执行 then 和 fi 之间的语句，否则显示 I love you。

4. 变量 go 被赋值为 0。程序控制将从 while 循环开始处开始，首先进行表达式测试。由于表达式值为假，循环退出，脚本从关键字 done 后的第 5 行语句开始执行。

5. done 标志循环体结束。

12.6.4 until 命令

用法和 while 命令相似，但检测退出状态时正好相反。until 命令检测紧随其后的命令表达式，如果其退出状态不为零，则执行循环体中的命令(do 和 done 之间的命令)。当到达 done 关键字后，控制返回到循环开始处，until 命令再次检查命令的退出状态，循环一直重复，直到 until 检测的退出状态为零，循环终止，转向执行 done 关键字以后的命令。

格式

```
until command
do
    command(s)
done
```

范例 12-36

```
#!/bin/ksh
1   until who | grep linda
2   do
        sleep 5
3   done
    talk linda@dragonwings
```

说明

1. until 循环检测管道中最后一条命令 grep 的退出状态。who 命令列出登录到这台机器上的用户名，并通过管道输出给 grep 命令。grep 命令只有在找到用户 linda 时，其退出状态值才为零(即成功退出)。

2. 如果用户 linda 未登录，进入循环体并沉睡眠 5 秒。

3. 如果用户 linda 已登录，grep 命令的退出状态为零，控制将跳转到 done 关键字后的语句。

范例 12-37

```
    #!/bin/ksh
1   hour=0
2   until (( hour > 23 ))
    do
```

```
3       case "$hour" in
        [0-9]|1[0-1]) print "Good morning!"
                ;;
       12) print "Lunch time"
                ;;
        1[3-7]) print "Siesta time"
                ;;
        *) print "Good night"
                ;;
        esac
4       (( hour+=1 ))
5   done
```

说明

1. 变量 hour 被赋值为零，此变量必须在 until 循环使用它之前初始化。

2. until 命令后是 let 命令。如果变量 hour 不大于 23，则退出状态为非 0，进入循环体。

3. case 命令将 hour 变量的值与所列的 hour 值进行匹配，如果匹配成功，就执行相应的语句。

4. hour 变量值加 1，否则，变量 hour 的值将永远小 23，循环则永远不会退出。控制返回到 until 语句，并将重新检测 hour 变量的值。

5. 关键字 done 标志着循环结束。当 hour 的值大于 23，控制将跳转到 done 之后的语句，如果 done 后没有语句，程序将终止。

12.6.5 select 命令和菜单

here 文档是生成菜单的简便方法，而 Korn shell 增加了一种新的循环，select 循环，它主要用于创建菜单。按数字顺序排列的菜单项将显示在标准错误输出上，并将显示 PS3 提示符请求用户输入(默认时，PS3 置为 #?符)。显示 PS3 提示符后，shell 将等待用户输入，输入的应当是菜单列表中的一项。输入值保存在一个 Korn shell 特殊变量 REPLY 中，它与选项列表中相应行的右括号前的字符串相关联[4]。

case 命令结合 select 命令一起使用，就能允许用户从菜单中进行选择，并基于选项执行相应的命令。LINES 和 COLUMNS 变量，将用来确定菜单在终端上的布局。其中，输出被显示到标准错误输出上，每一项的开头是一个数字和右括号，PS3 提示符显示在菜单底部。因为 select 命令是一条循环命令，因此，一定要记住用 break 命令或者 exit 命令退出循环。

格式

```
select var in wordlist
do
    command(s)
done
```

范例 12-38

(脚本)

---

④ 当循环再次执行时，如果希望菜单能够再次出现，则需要在 done 关键字前设置 REPLY 变量为空。

```
    #!/bin/ksh
    # Program name: goodboys
1   PS3="Please choose one of the three boys : "
2   select choice in tom dan guy
3   do
4       case $choice in
        tom)
            print Tom is a cool dude!
5           break;;         # break out of the select loop
6       dan | guy )
          print Dan and Guy are both sweethearts.
          break;;
        *)
7         print " $REPLY is not one of your choices" 1>&2
          print "Try again."
          ;;
8       esac
9   done
(命令行)
$ goodboys
1) tom
2) dan
3) guy
Please choose one of the three boys : 2
Dan and Guy are both sweethearts.
$ goodboys
1) tom
2) dan
3) guy
Please choose one of the three boys : 4
4 is not one of your choices
Try again.
Please choose one of the three boys : 1
Tom is a cool dude!
$
```

说明

1. PS3 变量被赋值为提示语句，出现菜单选项的下面。显示出提示语句后，程序开始等待用户输入。输入被存储在内置变量 REPLY 中。

2. select 命令后是变量 choice，其语法和 for 循环类似。变量 choice 被依次赋值为其后词列表中的元素，在本例中，即 tom，dan 和 guy。词列表将显示在菜单中，并以一个数字和右括号为前缀。

3. do 关键字表示循环体开始。

4. seletc 循环体中的第一条命令是 case 命令。case 经常与 select 命令合用。REPLY 变量中的值与某个选择相关联：1 和 tom 关联，2 和 dan 关联，3 和 guy 关联。

5. 如果选择了 tom，则先打印字符串“Tom is a cool dude!”，然后执行 break 命令退出 select 循环，程序控制从 done 后面的语句开始。



6. 如果用户选择了 2(dan)或者 3(tom)，REPLY 变量将保存用户的选择。

7. 如果选择的不是 1、2 或 3，一条错误消息将被发送到标准错误输出，并要求用户重试，控制转移到 select 循环开始处。

8. case 命令结束。

9. select 命令结束。

范例 12-39

（脚本）

```
    #!/bin/ksh
    # Program name: ttype
    # Purpose: set the terminal type
    # Author: Andy Admin
1   COLUMNS=60
2   LINES=1
3   PS3="Please enter the terminal type: "
4   select choice in wyse50 vt200 vt100 sun
    do
5       case $REPLY in
        1)
6           export TERM=$choice
            print "TERM=$choice"
            break;;                 # break out of the select loop
        2 | 3 )
           export TERM=$choice
           print "TERM=$choice"
           break;;
        4)
           export TERM=$choice
           print "TERM=$choice"
           break;;
        *)
7          print "$REPLY is not a valid choice. Try again" 1>&2
               ;;
        esac
8  done
```

（命令行）

```
$ ttype
1) wyse50   2) vt200   3) vt100   4) sun
Please enter the terminal type : 4
TERM=sun

$ ttype
1) wyse50   2) vt200   3) vt100   4) sun
Please enter the terminal type : 3
TERM=vt100

$ ttype
1) wyse50   2) vt200   3) vt100   4) sun
Please enter the terminal type : 7
7 is not a valid choice. Try again.
```

```
Please enter the terminal type: 2
TERM=vt200
```

说明

1. COLUMNS 变量被设置为菜单的宽度，即 select 循环生成的菜单在终端显示上的列数。默认值为 80。

2. LINES 变量控制菜单在终端显示时垂直方向上的行数，默认为 24 行。如果把 LINES 变量的值置为 1，菜单项将打印在一行上，而不是如上例所示的垂直显示。

3. 设置 PS3 提示符，该提示符的内容将显示在菜单选项下。

4. select 循环打印出一个有 4 个选项的菜单：wyse50、vt200、vt100 和 sun。根据变量 REPLY 中保存的用户响应值，choice 将被赋值为以上所列的某个值。如果 REPLY 为 1，将 wyse50 赋给 choice；如果 REPLY 为 2，将 vt200 赋给 choice；如果 REPLY 是 3，将 vt100 赋给 choice；如果 REPLY 是 4，将 sun 赋给 choice。

5. REPLY 变量值等于用户输入的选择。

6. 给终端类型赋值，并将该变量导出、打印。

7. 如果用户没有输入 1 和 4 之间的数字，用户将被提示重新输入。要注意的是，此时将仅显示 PS3 提示，而不显示菜单。要重新显示菜单，必须将 REPLY 值设为空(null)，即在第 8 行上输入：REPLY=。

8. select 循环终止。

12.6.6 循环控制命令

在一些情况下，您可能想跳出一个循环并返回循环开始处，或是想终止一个无限循环。Korn shell 提供了循环控制命令来控制循环。

shift 命令 shift 命令把参数列表向左移动一定的次数，不带参数时，是向左移动一次。一旦移动列表，左边的参数将被永远移出。shift 命令通常是在 while 循环中，遍历整个位置参量列表时使用。

格式

```
shift [n]
```

范例 12-40

(无循环)

(脚本)

```
   #!/bin/ksh
   # Scriptname: doit0
1  set joe mary tom sam
2  shift
3  print $*
4  set $(date)
5  print $*
6  shift 5
7  print $*
8  shift 2
```

(输出)




```
    $ doit0
3   mary tom sam
5   Sun Sep 9 10:00:12 PDT 2004
7   2004
8   ksh: shift: bad number
```

说明

1. 用 set 命令设置位置参量，$1 被赋值为 joe，$2 被赋值为 mary，$3 被赋值为 tom，$4 被赋值为 sam。

2. shift 命令向左移动位置参量，joe 被移出。

3. 移动后打印参数列表。$* 代表所有的参数。

4. 用 set 命令重新设置位置参量，设为 data 命令的输出。

5. 打印新的参数表。

6. 把参数表向左移动 5 次。

7. 打印新的参数表。

8. 在尝试移动的次数多于参数的个数时，shell 将发送一条消息到标准错误输出。

范例 12-41

(循环)
(脚本)

```
    #!/bin/ksh
    # Usage: doit [args]
1   while (( $# > 0 ))
    do
2      print $*
       shift
4   done
```

(命令行)

```
$ doit a b c d e
a b c d e
b c d e
c d e
d e
e
```

说明

1. while 命令检测数字表达，如果位置参量的个数($#)大于零，则进入循环体。位置参量来自于命令行参数，共有 5 个。

2. 打印所有的位置参量。

3. 把参数表向左移动一次。

4. 循环体在此结束，控制返回到循环开始处。参数列表每次被减 1。在第一次左移之后，$# 为 4。当 $# 减为 0 时，循环结束。

break 命令　内置的 break 命令用于强行退出循环，而不是退出程序(用 exit 命令退出程序)。break 命令执行后，控制转移到 done 关键字后开始。break 命令退出最内层的循环，如果是在嵌套循环中，break 命令可以带一个数字作为参数，以退出任意层数的外部循环。使用 break 命令，可以方便地从无限循环中跳出。

格式

```
break [n]
```

范例 12-42

```
1  while true; do
2      read answer? Are you ready to move on\?
3      if [[ $answer = [Yy]* ]]; then
           break
4      else
               ....commands....
       if
5  done
6  print "Here we are"
```

说明

1. true 命令是一个 UNIX 命令，是 Korn shell 中冒号命令的别名，总是以状态 0 退出，常被用来启动一个无限循环(冒号命令，即空命令也起类似作用)。这里将进入循环体。

2. 要求用户输入，输入值被赋给变量 answer。

3. 如果 answer 的值为任一以 Y 或 y 开头的单词，如 Y、y、Yes、Yup 或 Ya 等，将执行 break 命令，并将控制跳转到第 6 行，屏幕上将显示“Here we are”。如果用户的输入不是以 y 或 Y 开头的单词，程序就一直请求用户输入。

4. 如果第 3 行的测试失败，将执行 else 命令。当循环体在 done 关键字处结束时，控制再次转向第 1 行 while 循环的起始处。

5. 循环体结束。

6. break 命令执行后，控制从这里开始。

continue 命令 在某些条件满足时，continue 命令使控制返回循环起始处开始执行。循环体中 continue 之后的命令将被忽略。continue 语句使控制返回到最内层的循环起始处。在嵌套循环中，continue 语句可带一个数字作为参数，以使控制精确地退回到任意一个外部循环的起始处。

格式

```
continue [n]
```

范例 12-43

(邮件列表)

```
$ cat mail_list
ernie
john
richard
melanie
greg
robin
```

(脚本)

```
# Scriptname: mailtomaillist
#!/bin/ksh
```



```
1   for name in $(<mail_list)
    do
2      if[["$name"="richard"]]then
3           continue
       else
4           mail $name < memo
      fi
5   done
```

说明

1. for 循环从 mail_list 文件中依次读取所列的名字。每次循环把列表中的一个名字赋给变量 name，然后把这个名字从列表中移出，接着用列表中的下一个名字来赋值。

2. 名字匹配了 richard，则执行 continue 语句。移出 richard 后，变量 name 将被赋值为下一个用户名 melanie。

3. continue 语句将控制返回到循环起始处，而忽略循环体中剩余的命令。

4. 列表中除了 richard 外的所有用户，都将被邮寄一份 memo 文件的副本。

5. 循环体结束。

12.6.7 嵌套循环和循环控制

在嵌套循环中，break 和 continue 命令可用来终止循环。

范例 12-44

(脚本)

```
    #!/bin/ksh
1   while true ; do
             < Commands here>
2     for user in tom dick harry joe
      do
        if [[$user = [Dd]*]]
        then
3         continue 2
          <Commands here>
4         while true
          do
              <Commands here>
5           break 3
6           done
7     fi
     done
8   done
9   print Out of loop
```

说明

1. true 命令总是返回一个退出状态值 0，因此循环成为一个无限循环，除非使用循环控制命令来终止它。

2. 进入 for 循环。

3．for 循环将依次读取列表中的每个名字。如果变量 user 是以 d 或 D 开头，则执行 continue 语句，将控制转移到 while 循环的起始处。不带参数的 continue 命令使控制从 for 循环起始处开始。参数 2 告诉 shell 跳到第 2 个封闭的循环起始处，并重新开始执行。

4．while 循环被嵌套，true 命令总是以零状态退出，循环将一直执行下去。

5．break 命令终止最外层的 while 循环。从第 9 行开始执行。

6．done 关键字标志着最内层的 while 循环结束。

7．done 关键字标志 for 循环的结束。

8．done 关键字标志着最外层的 while 循环结束。

9．循环体外部。

12.6.8 I/O 重定向和循环

Korn shell 允许在循环中应用重定向和管道。与 Bourne shell 不同的是，循环是在当前 shell 中运行，而不是在派生的子 shell 中运行。而且当循环结束时，在循环中设置的变量保持不变。

将循环输出重定向到一个文件 除了输出到屏幕外，循环的输出还可以被重定向到文件或管道。参见范例 12-45。

范例 12-45

```
(命令行)
1    $ cat memo
     abc
     def
     ghi
--------------------------------------------------------------------
(脚本)
     #!/bin/ksh
     # Program name: numberit
     # Put line numbers on all lines of memo
2    if (( $# < 1 ))
     then
         print "Usage: $0 filename " >&2
         exit 1
     fi
3    integer count=1              # Initialize count
4    cat $1 | while read line     # Input is coming from memo
     do
5       (( count == 1 )) && print "Processing file $1..." > /dev/tty
6       print $count $line
7       (( count+=1 ))
8   done > tmp$$                  # Output is going to a temporary file
9   mv tmp$$ $1
(命令行)
10   $ numberit memo
     Processing file memo...
11   $ cat memo
```



```
1 abc
2 def
3 ghi
```

说明

1. 显示文件 memo 的内容。

2. 如果参数的个数小于 1，标准错误输出(屏幕)上将显示一条信息，说明使用方法。

3. count 变量被声明为整型变量，并被赋值为 1。

4. UNIX 命令 cat 显示一个文件的内容，该文件的文件名保存在变量 $1 中。把 cat 命令的输出用管道传送给 while 循环。在第一次循环中，read 命令读取文件的第一行并赋给变量 line。第二次循环，读取文件的第二行赋给变量 line，以此类推。

5. print 语句的输出被送到 /dev/tty，即屏幕。如果没有被明确的重定向到 /de/tty，输出将在第 8 行被重定向到临时文件 tmp$$。

6. 打印 count 变量的值和文件中的行。

7. count 变量加 1。

8. 除了第 3 行以外，整个循环的输出被重定向到文件 tmp$$($$ 为此进程的 PID)。把进程 PID 号附加到文件名末尾，以使临时文件有一个唯一的名字。

9. 临时文件被重新命名为变量 $1 中保存的文件名。

10. 执行程序，程序将对文件 memo 进行处理。

11. 显示文件内容，看到每一行都增加了一个行号。

将循环输出用管道送到一个 UNIX 命令中　循环的输出可以从屏幕重定向到一个管道。参见范例 12-46。

范例 12-46

```
(脚本)
1    for i in 7 9 2 3 4 5
2    do
        print $i
3    done | sort -n
(输出)
2
3
4
5
7
9
```

说明

1. for 循环遍历未排序数字列表中的数字。

2. 循环体中，执行打印数字操作。输出将通过管道送给 UNIX 的命令 sort。

3. 在 done 关键字之后创建一个管道。

12.6.9 在后台运行循环

如果循环需要一定的时间来处理，则可以将循环置于后台运行，这样前台可以继续运行其他程序。

范例 12-47

```
1  for person in bob jim joe sam
   do
2     mail $person < memo
3  done &
```

说明

1. for 循环依次读取列表中的每一个名字：bob，jim，joe 和 sam，依次将每个名字赋值给变量 person。

2. 在循环体内，向每个人发送一封包含 memo 文件内容的邮件。

3. done 关键字后的&符，使得循环在后台运行期间。在循环执行期间，前台可以继续运行其他程序。

12.6.10 exec 命令和循环

exec 命令可用来在不创建子 shell 的情况下，关闭标准输入或标准输出。

范例 12-48

```
(文件)
1   cat tmp
    apples
    pears
    bananas
    peaches
    plums
------------------------------------------------------------
(脚本)
    #!/bin/ksh
    # Scriptname: speller
    # Purpose: Check and fix spelling errors in a file
2   exec < tmp           # Opens the tmp file
3   while read line      # Read from the tmp file
    do
4      print $line
5      print -n "Is this word correct? [Y/N] "
6      read answer < /dev/tty    # Read from the terminal
       case $answer in
       [Yy]*)
          continue
             ;;
       *)
          print "New word? "
7         read word < /dev/tty
          sed "s/$line/$word/" tmp > error
          mv error tmp
8         print $word has been changed.
             ;;
       esac
    done
```



说明

1. 显示文件 tmp 的内容。

2. exec 命令改变了标准输入(文件描述符 0)的设置。输入将来自于文件 tmp，而不是键盘。

3. 启动 while 循环，read 命令从 tmp 文件读入一行。

4. 将变量 line 的值显示在屏幕上。

5. 询问用户，上面的单词是否正确。

6. read 命令从终端 /dev/tty 中得到用户响应。如果不将输入重定向到从终端读取，则会继续从 tmp 文件读取输入。

7. 再次要求用户重新输入，输入被重定向到从终端 /dev/tty 读取。

8. 显示新单词。

12.6.11 IFS 和循环

IFS 是 shell 的内部字段分隔符，可以是空格、制表符和换行符。当一条 UNIX 命令(如：read、set、for 和 select)需对一长串单词进行解析时，IFS 就用来作为单词(token)之间的分隔符。如果要在一个单词列表中使用不同的分隔符，用户可以进行重置。用户在重新设置分隔符之前，最好将其原始值保存到另外一个变量中去，这样，便于取回其默认值。

范例 12-49

(脚本)

```
    #!/bin/ksh
    # Scriptname: runit
    # IFS is the internal field separator and defaults to
    # spaces, tabs, and newlines.
    # In this script it is changed to a colon.
1   names=Tom:Dick:Harry:John
2   OLDIFS="$IFS"          # Save the original value of IFS
3   IFS=":"
4   for persons in $names
    do
5      print Hi $persons
    done
6   IFS="$OLDIFS"            # Reset the IFS to old value
7   set Jill Jane Jolene     # Set positional parameters
8   for girl in $*
    do
       print Howdy $girl
    done
```

(输出)

```
$ runit
Hi Tom
Hi Dick
Hi Harry
Hi John
```

```
Howdy Jill
Howdy Jane
Howdy Jolene
```

说明

1. 变量 names 被设置为字符串“Tom:Dick:Harry:John”，单词之间用冒号分隔。

2. 将 IFS 的值赋给另一个变量 OLDIFS。由于 IFS 变量的值为空格符，因此必须用双引号来保护它。

3. IFS 被赋值为冒号“:”，现在冒号被用来分隔单词。

4. 在变量替换后，for 循环将从名字列表中依次读取单词，其中的内部字段分隔符是冒号。

5. 显示单词列表中的每个名字。

6. IFS 的原始值存储在 OLDIFS 变量中，将此值重新赋给 IFS。

7. 设置位置参量。$1 被设置为 Jill，$2 被设置为 Jane，$3 被设置为 Jolene。

8. $* 代表所有的位置参量：Jill、Jane 和 Jolene。for 循环在每次递归中，会将这些名字依次赋给变量 girl。

12.7 数组

Korn shell 数组是一个一维数组，最大可包含 1024 个单元(大小可变)，一个单元可以是一个单词或整数。数组下标从 0 开始。数组中的每个单元都可以单独赋值或重置，数组元素的赋值没有什么顺序要求。例如，可以先给数组的第 10 个元素赋值，再给第 1 个元素赋值。可以用带 - A 选项的 set 命令给一个数组赋值。

1988 年以后版本的 Korn Shell 开始支持关联数组。

范例 12-50

(命令行)

```
1    $ array[0]=tom
     $ array[1]=dan
     $ array[2]=bill
2    $ print ${array[0]}     # Curly braces are required.
    tom
3    $ print ${array[1]}
     dan
4    $ print ${array[2]}
     bill
5    $ print ${array[*]}     # Display all elements.
     tom dan bill
6    $ print ${#array[*]}    # Display the number of elements.
     3
```

说明

1. 对数组的前 3 个元素赋值。数组下标从 0 开始。



2. 显示第 1 个数组元素的值：tom。注意，必须用花括号把变量括起来，否则$array[0]的输出结果是 tom[0]。

3. 显示第 2 个数组元素的值：dan。

4. 显示第 3 个数组元素的值：bill。

5. 显示数组中所有元素的值。

6. 显示数组的元素个数。如果知道了大小和类型，可以用 typeset 来声明一个数组。

范例 12-51

```
(At The Command Line)
1    $ typeset -i ints[4]        # Declare an array of four integers.
2    $ ints[0]=50
     $ ints[1]=75
     $ ints[2]=100
3    $ ints[3]=happy
     ksh: happy: bad number
```

说明

1. 用命令 typeset 创建一个包含 4 个整数的数组。
2. 给数组元素赋整数值。
3. 对数组的第 4 个元素赋字符串值，则 Korn Shell 向标准错误输出发送一条出错信息。

用 set 命令创建数组

您可以用 set 命令为一个数组的元素赋值。命令行中 - A 选项之后的第一个词是数组的名称，其余的词则是数组元素的值。

范例 12-52

```
(命令行)
1   $ set -A fruit apples pears peaches
2   $ print ${fruit[0]}
    apples
3   $ print ${fruit[*]}
    apples pears peaches
4   $ fruit[1]=plums
5   $ print ${fruit[*]}
    apples plums peaches
```

说明

1. 带 - A 选项的 set 命令创建了一个数组。- A 选项后的是数组的名字 fruit，数组名之后的是数组的各个元素。

2. 数组下标从 0 开始。必须用花括号把变量括起来，这样才能够正确地计算变量的值。数组的第 1 个元素被显示在屏幕上。

3. 当星号用作下标时，将显示数组的所有元素。

4. 将数组的第 2 个元素重新赋值为 plums。

5. 显示数组的所有元素。

12.8 函数

Korn Shell 函数与那些在 Bourne Shell 中使用的函数类似，用于实现模块化的程序。一个函数是一条或多条命令的集合，只要输入函数名，就可以执行这些命令，像执行内置的 shell 命令一样。下面是一些使用函数的重要规则。

(1) Korn Shell 首先执行内置的命令，然后是函数，最后是其他可执行程序。函数在定义时会被读入内存一次，以后每次引用函数时不再读入。

(2) 一个函数必须在使用前定义。因此，最好是在脚本的开头就定义函数。

(3) 函数在脚本的当前环境下运行，函数和调用它的脚本共享变量，并且允许您通过设置位置参量来传递参数。函数的当前工作目录是脚本调用函数时的工作目录。如果在函数中改变了目录，则脚本的工作目录也同时被改变。

(4) 在 Korn Shell 中，可以用 typeset 命令在函数中声明局部变量。ksh 函数可以被导出到子 shell 中。

(5) return 语句返回函数中最后一条执行的命令的退出状态值，或者返回一个指定的参数值。返回值不能大于 255。

(6) 使用预设的命令别名 functions，可以列出所有的函数和函数定义。

(7) 陷阱(Trap)对函数而言是局部的，函数中设置的 trap，在函数退出时，将恢复为原先的值(Bourne shell 中则不同)。

(8) 函数可以递归，也就是调用它们本身。要小心地处理递归，否则，Korn Shell 会发出警告“嵌套太深了”。

(9) 函数可以被自动载入内存，这使函数只有在被引用到的时候才真正定义。如果一个函数永远不被引用，它就不会被载入到内存中。

(10) 1988 年以后版本的 Korn Shell 支持规程函数(discipline functions)，并通过引用以及复合变量来传递参数。将不会再出现想执行某个函数的时候却执行同名 shell 内置命令的情况。而在老的版本中，必须使用别名和函数的组合来写一个函数，以覆盖同名的内置命令[⑤]。

12.8.1 定义函数

一个函数必须在调用之前被定义。Korn shell 通过在函数名前加上关键字 function 来定义函数。在每个花括号的内侧必须都有一个空格(老式的Bourne shell 函数定义参见 12.8 “函数”一节，Korn shell 也同时兼容这些定义)。

格式

```
变量 变量名 { 命令; 命令; }
```

范例 12-53

```
function usage { print "Usage $0 [-y] [-g] " ; exit 1; }
```

⑤ 参见 David G. Korn 和 Morris I. Bolsky 编著的 *The Korn Shell Command and Programming Language*。



说明

函数名是 usage。如果脚本没有接收到正确的参数 -y 或者 -g，则打印一个诊断信息，并退出脚本。

12.8.2 列出和取消函数定义

命令 typeset –f 列出局部的函数定义。命令 typeset –fx 列出被导出的函数定义。命令 unset –f function_name 用于取消一个函数的定义。参见表 12-11。

表 12-11 typeset 命令和功能选项

选 项	功 能
typeset –f	显示所有函数及其定义。必须有一个历史文件，用来保存所有的函数定义
typeset +f	只显示函数名称
typeset –fx	与要创建一个独立的 ksh 副本不同，有一类函数能从一个 shell 脚本导出到另一脚本，显示此类函数的定义
typeset –fu func	func 是一个尚未被定义的函数名

12.8.3 局部变量和返回值

命令 typeset 可用于创建局部变量。这些变量只在创建它们的函数中有效，一旦到了函数的外部，这些变量就是未定义的。

函数的返回值是函数内最后一条命令执行后的退出状态值，或是 return 命令指定的返回值。如果为 return 命令指定了一个值，那么这个值保存在变量 ? 里。返回值可以取 0~255 之间的整数。由于 return 命令被限制为只能返回整数值，可以使用命令替换来返回函数的输出，并且将函数的输出赋给一个变量，就像是获得一个 UNIX 命令的输出一样。

范例 12-54

(脚本)

```
    #!/bin/ksh
    # Scriptname: do_increment
    # Using the return Command)
1   function increment {
2     typeset sum        # sum is a local variable.
      ((sum = $1+1))
3     return $sum        # Return the value of sum to the script.
    }

    print -n "The sum is"
4   increment 5          # Call the function increment and pass 5 as a
                         # parameter. 5 becomes $1 for the increment function.
5   print $?             # The return value is stored in the ? variable
6   print $sum           # The variable "sum" was local to the function, and
                         # is undefined in the main script. Nothing is printed.
```

(输出)

```
    $ do_increment
```

```
5    The sum is 6
     6
```

说明

1. 定义函数 increment。
2. 命令 typeset 为函数定义局部变量 sum。
3. 当给内置命令 return 一个参数时，参数将保存在变量 ? 中，并将返回到脚本主流程中，函数被调用所在的那一行之后。在本脚本中，调用函数 increment 时带了一个参数。
4. 调用函数 increment 时带了参数 5。
5. 除非是显式地为 return 命令设置了一个参数，否则，变量 ? 中将保存函数的退出状态。return 命令的参数指定了函数的返回状态，并且其值保存在变量 ? 中，取值范围必须在 0~255 之间。
6. 既然 sum 是被定义成函数 increment 的一个局部变量，那么在脚本中(函数外部)它就是未被定义的。在函数之外输出 sum 的值将是空值。

范例 12-55

(使用命令替换)

(脚本)

```
    # Scriptname: do_square
    #!/bin/ksh
1   function square {
     (( sq = $1 * $1 ))
       print "Number to be squared is $1."
2      print "The result is $sq "
    }
3   read number?"Give me a number to square. "
4   value_returned=$(square $number)
5   print $value_returned
```

(输出)

```
    $ do_square
5   Number to be squared is 10. The result is 100
```

说明

1. 定义函数 square。函数的功能是计算参数的平方值。
2. 打印平方运算的结果。
3. 要求用户输入数字。
4. 带一个数字(用户输入的)作为参数调用函数 square。函数被封装在前面带有 $ 符的括号中，因此，这里作了命令替换。函数的输出(包括函数的两条打印语句)赋给了变量 value_returned。
5. 命令替换删去了字符串“Number to be squared is”和“The result is 100”之间的换行符。



12.8.4 导出函数

除非是用 typeset 命令在 ENV 文件中定义了函数，如：typeset –fx function_names，否则，子 shell 不会继承任何函数定义。

可以用 typeset –fx 命令从当前的 Korn shell 导出函数给一个脚本，或者是从一个脚本导出到另一个脚本，但是不能从一个 ksh 的实例导出函数到另一个 ksh 实例(一个实例是指当您在提示符下输入 ksh 时，将启动一个新的 shell)，新 shell 不会继承导出的函数定义。

范例 12-56

(第一个脚本)

```
    $ cat calling_script
    #!/bin/ksh
1   function sayit { print "How are ya $1?" ; }
2   typeset -fx sayit   # Export sayit to other scripts
3   sayit Tommy
4   print "Going to other script"
5   other_script        # Call other_script
    print "Back in calling script"
*******************************************************************
```

(第二个脚本)

```
    $ cat other_script    # NOTE: This script cannot be invoked with
                          #!/bin/ksh
6   print "In other script "
7   sayit Dan
8   print "Returning to calling script"
```

(输出)

```
    $ calling_script
3   How are ya Tommy?
4   Going to other script
6   In other script
7   How are ya Dan?
8   Returning to calling script
    Back in calling script
```

说明

1. 定义函数 sayit。函数接受一个保存在变量 $1 中的参数。
2. 带 - fx 选项的 typeset 命令允许函数被导出给脚本中调用的任何其他脚本。
3. 带参数 Tommy 调用函数 sayit。在函数中，Tommy 将保存在变量 $1 中。
4. 函数 sayit 结束后，脚本程序从这里继续运行。
5. 执行名为 other_script 的脚本。
6. 现在是在另一个脚本中，该脚本被第一个脚本调用。该脚本不能以行“#!/bin/ksh”打头，因为这一行会启动一个 ksh 子 shell，也就是调用一个独立的 Korn shell，这样导出的函数就不起作用了。
7. 调用函数 sayit。Dan 作为参数被传递，将保存在函数中的$1 中。
8. 输出了这一行后，脚本 other_script 结束，控制返回到调用 other_script 的脚本中，脚本 other_script 被调用时所在位置的下一行。

12.8.5 typeset 命令和函数选项

typeset 命令用于显示函数的属性。参见表 12-11。

自动加载函数 一个自动加载函数，只有在被引用时才加载到程序中。自动加载函数可以在其他目录下的文件中定义，而不一定要在脚本中定义，这样可以使脚本简单紧凑。要自动加载函数，需要在 ENV 文件中设置 FPATH 变量。FPATH 变量包含了一个搜索路径，这些路径的目录下包含定义函数的文件，且要求文件与该文件中定义的函数同名。

命令 typeset –fu 的别名是 autoload，该命令将尚未被定义的函数作为自动加载函数。当执行了带一个函数名作为参数的 autoload 命令之后，必须通过调用函数来执行函数里的命令。而自动加载函数最主要的优势是提供更好的性能，因为 Korn shell 如果无需引用某个函数，就不需要读入该函数的定义[6]。

范例 12-57

```
(命令行)
1   $ mkdir functionlibrary
2   $ cd functionlibrary
3   $ vi foobar
(在编辑器)
4   function foobar { pwd; ls; whoami; }  # function has the same name as
                                          # the file.
(.profile 文件)
5  export FPATH=$HOME/functionlibrary  # This path is searched for functions.
(在脚本)
6  autoload foobar
7  foobar
```

说明

1. 创建一个目录来保存函数。
2. 进入这个目录。
3. foobar 是目录 functionlibrary 下的一个文件。文件 foobar 包含了函数 foobar 的定义，文件名和函数名必须相同。
4. 在文件 foobar 中定义函数 foobar。
5. 在用户的初始化文件.profile 中，把函数定义文件所在的路径赋给 FPATH 变量。当自动加载函数时，Korn shell 在这个路径中搜索函数。FPATH 变量被导出。
6. 在脚本中，函数 foobar 被载入程序的内存空间。
7. 调用函数 foobar。

可以在一个文件中定义多个函数，例如，各种算术函数可以定义在一个名为 math 的文件中。由于自动加载规定，函数必须与定义它的文件同名，因此就要为每个函数分别建立指向函数文件的强制链接。可以以每个函数名作为一个链接，指向定义这个函数的文件。例如，如果 math 文件中有个名为 square 的函数，则使用 UNIX 命令 ln 给 math 文件赋予另

⑥ Morris I. Bolsky 和 David G. Korn 所编著的 *The New Kornshell*。



一个文件名 square。现在文件 math 和 square 都可以被引用，并且可以通过对应的函数名来分别引用这两个文件。现在函数 square 可以通过它自己的名字来自动加载了。

范例 12-58

（命令行）

```
1    $ ln math square add divide
2    $ ls -i
     12256 add
     12256 math
     12256 square
     12256 divide
3    $ autoload square; square
```

说明

1. UNIX/Linux 的 ln(链接)命令使一个文件可以有不同的名称。文件 math 和 square 指的是同一个文件。每创建一个链接，文件的链接数加 1。

2. 如果一个文件列表表明，所有的文件有同样的 inode 号，说明它们其实是同一个文件，但可以通过不同的名称来访问。

3. 当自动加载文件 square 时，函数 square 与文件同名，因此将调用 square 函数，而文件中定义的其他函数不会被引用，如果要引用这些函数，就必须加载与这些函数同名的文件。

12.9 trap 命令

当程序运行时，如果按下 Ctrl+C 或 Ctrl+\组合键，程序接收到信号后会立刻结束。有的时候您可能不希望信号到达的时候程序立刻结束，或者希望在真正退出脚本之前执行一系列的清除操作，trap 命令允许控制程序收到信号以后的行为。

信号是一种异步消息，用一个数字表示，当按下某个键或者出现异常事件时，该数字可以从一个进程发送给另一个进程，或者由操作系统发送给进程[7]。trap 命令告知 shell，在收到信号以后立即结束正在运行的命令。如果 trap 命令后面紧跟着用单引号括起来的若干条命令，则在接收到特定的信号后将会执行这些命令。使用命令 kill –l 可以得到一个关于所有信号及其对应数字的列表。

格式

```
trap 'command; command' signal
```

范例 12-59

```
trap 'rm tmp*$$; exit 1' 1 2 15
```

说明

当信号 1(挂起)、2(中断)、或 15(软件终止)中的任何一个信号到达时，删除所有的临时

⑦ 可参考 Morris I. Bolsky 和 David G. Korn 编著的 *The New KornShell Command and Programming Language*。

文件，然后退出。

如果在脚本运行时发生一个中断，trap 命令允许您以多种方式处理中断。你可以按通常的方式处理信号(默认方式)、忽略信号或是创建一个处理函数以在信号到来时调用。表 12-12 列出了信号编号和它们对应的名称[8]。输入 kill -l 可以得到如表 12-12 所示的输出。

表 12-12 信号[9] [10]

1) HUP	12) SYS	23) POLL
2) INT	13) PIPE	24) XCPU
3) QUIT	14) ALRM	25) XFSZ
4) ILL	15) TERM	26) VTALRM
5) TRAP	16) URG	27) PROF
6) IOT	17) STOP	28) WINCH
7) EMT	18) TSTP	29) LOST
8) FPE	19) CONT	30) USR1
9) KILL	20) CHLD	31) USR2
10) BUS	21) TTIN	
11) SEGV	22) TTOU	

12.9.1 伪信号

有 3 种伪信号，它们不是真正的信号，shell 产生这些信号是用来帮助调试程序。而 trap 命令则像对待真正的信号一样来处理它们，其定义的方式也相同。表 12-13 中列出了伪信号。

表 12-13 Korn shell 的伪陷入信号

信　号	功　能
DEBUG	在每一个脚本命令后执行 trap 命令
ERR	如果脚本中有任何命令返回一个非 0 的退出状态值，就执行 trap 命令
0 或 EXIT	如果 shell 退出，就执行 trap 命令

HUP 和 INT 这些信号的名字之前通常加上前缀 SIG，例如 SIGHUP、SIGINT 等。Korn shell 允许使用没有 SIG 前缀的符号名来表示信号，或者是信号的编号。参见后面的范例 12-60。

12.9.2 复位信号

要复位某个信号的处理方式，使之恢复默认的行为，使用 trap 命令，后跟信号的名字或编号即可。函数中设置的陷入仅限于函数局部，也就是说，它们在函数外部是无效的。

---

⑧ 想要详细了解 UNXI 信号，参见 W. Richard Stevens 编著的 Advanced Programming in the UNIX Environment。

⑨ 本命令的输出会随操作系统的不同而有一些不同。

⑩ 需要了解 UNIX 信号和其含义的完整信息，可以访问 www.cybermagician.co.uk/technet/unixsignals.htm。需要了解 Linux 信号，可以访问 www.comptechdoc.org/os/linux/programming/linux_pgsignals.html。



范例 12-60

```
trap 2 or trap INT
```

说明

复位信号 2、SIGINT，使之恢复默认行为，它的默认动作是当按下中断键(Ctrl+C)时，终止进程。

12.9.3 忽略信号

如果 trap 命令后跟一对空的双引号，命令中列出的信号将被进程忽略。

范例 12-61

```
trap " " 1 2    or    trap "" HUP INT
```

说明

信号 1(SIGHUP)和 2(SIGINT)将被 shell 进程忽略。

12.9.4 列出信号

只是输入 trap 命令，将列出所有的陷阱设置和处理陷阱的命令。

范例 12-62

(脚本)

```
    #!/bin/ksh
    # Scriptname: trapping
    # Script to illustrate the trap command and signals
    # Can use the signal numbers or ksh abbreviations seen
    # below. Cannot use SIGINT, SIGQUIT, etc.
1   trap 'print "Ctrl-C will not terminate $PROGRAM."' INT
2   trap 'print "Ctrl-\ will not terminate $PROGRAM."' QUIT
3   trap 'print "Ctrl-Z will not terminate $PROGRAM."' TSTP
4   print "Enter any string after the prompt.\
    When you are ready to exit, type \"stop\"."
5   while true
    do
6       print -n "Go ahead...> "
7       read
8       if [[ $REPLY = [Ss]top ]]
        then
9           break
        fi
10  done
```

(输出)

```
    $ trapping
4   Enter any string after the prompt.
    When you are ready to exit, type "stop".
6   Go ahead...> this is it^C
1   Ctrl-C will not terminate trapping.
6   Go ahead...> this is it again^Z
```

```
3    Ctrl-Z will not terminate trapping.
6    Go ahead...> this is never it^\
2    Ctrl-\ will not terminate trapping.
6    Go ahead...> stop
     $
```

说明

1. 第 1 个 trap 命令捕捉 INT 信号，即 Ctrl-C。如果在程序运行时按下 Ctrl+C 组合键，将执行在单引号中包括的命令。程序将不终止，而是显示信息“Ctrl+C will not terminate trapping”，并且继续提示用户输入。

2. 第 2 个 trap 命令将在用户按下 Ctrl+\组合键或产生 QUIT 信号时执行。打印信息“Ctrl - \ will not terminate trapping”，程序继续运行。默认情况下，信号 SIGQUIT 将终止进程，并产生一个 core 文件。

3. 第 3 个 trap 命令将在用户按下 Ctrl+Z 组合键或产生 TSTP 信号时执行。打印信息“Ctrl - Z will not terminate trapping”，程序继续运行。如果系统实现了作业控制的功能，这个信号通常是让程序在后台挂起。

4. 提示用户输入。

5. 进入 while 循环。

6. 打印字符串“Go ahead …>”，程序等待用户输入(见下一行的 read)。

7. read 命令将用户的输入赋给内置的 REPLY 变量。

8. 如果 REPLY 的值匹配字符串“Stop”或“stop”，则执行 break 命令，使循环退出，程序也将结束。输入“Stop”或“stop”是我们退出程序的唯一方式，也可以用 kill 命令终止程序。

9. break 命令实现从循环体中退出。

10. 关键字 done 标志着循环结束。

范例 12-63

(脚本)

```
     $ cat trap.err
     #!/bin/ksh
     # This trap checks for any command that exits with a nonzero
     # status and then prints the message.
1    trap 'print "You gave me a non-integer. Try again. "' ERR
2    typeset -i number     # Assignment to number must be integer
3    while true
     do
4       print -n "Enter an integer. "
5       read -r number 2> /dev/null
6       if (( $? == 0 ))      # Was an integer read in?
        then                  # Was the exit status zero?
7            break
          fi
     done
8    trap-ERR                  # Unset pseudo trap for ERR
     n=$number
```




```
9    if grep ZOMBIE /etc/passwd > /dev/null 2>&1
     then
        :
     else
10      print "\$n is $n. So long"
     fi
```

(输出)

```
     $ trap.err
4    Enter an integer. hello
1    You gave me a non-integer. Try again.
4    Enter an integer. good-bye
1    You gave me a non-integer. Try again.
4    Enter an integer. \\\
1    You gave me a non-integer. Try again.
4    Enter an integer. 5
10   $n is 5. So long.
     $ trap.err
4    Enter an integer. 4.5
10   $n is 4. So long.
```

说明

1. 每当程序中的命令返回一个非 0 的退出状态，也就是执行失败时，ERR(伪)信号就打印双引号中的消息。

2. 带 - i 选项的 typeset 命令创建一个整型变量 number，这个变量只能赋整数值。

3. true 命令的退出值总是 0。进入 while 循环体。

4. 要求用户输入一个整数。

5. read 命令读取用户的输入，然后赋给 number 变量。输入的数字必须是一个整数。如果不是，程序将向 /dev/null 发送一条错误消息。read 命令的 - r 选项允许您输入一个负数(以减号开头)。

6. 如果 read 命令的退出状态是 0，即成功地输入了一个数，则执行 if 语句。

7. 执行 break 命令，退出循环。

8. 取消对 Korn shell 伪信号 ERR 的陷入处理。

9. 当 grep 语句执行失败时，它返回一个非 0 的退出状态。如果我们没有取消 ERR 的陷入处理，脚本程序就会打印 “You gave me a non - integer. Try again.”。只要 grep 命令在 /etc/passwd 文件中找不到 ZOMBIE，就会一直打印这样的信息。

10. 如果 grep 命令失败，则打印这一行。注意，如果输入了一个浮点数，如 4.5，则数字将被取整。

12.9.5 陷入和函数

如果 trap 用在一个函数中，trap 设置和它的处理命令都是仅限于函数内部，仅在函数内有效。

范例 12-64

(脚本)

```
    #!/bin/ksh
1   function trapper {
        print "In trapper"
2       trap 'print "Caught in a trap!"' INT
        print "Got here."
        sleep 25
    }
3   while :
    do
4      print "In the main script"
       trapper   # Call the function
5      print "Still in main"
       sleep 5
       print "Bye"
    done
------------------------------------------------------
(输出)
$ functrap
In the main script
In trapper
Got here.
^CCaught in a trap!
$
```

说明

1. 定义函数 trapper，函数中包含 trap 命令。

2. 如果按下 Ctrl+C 组合键将执行 trap 命令。执行 trap 中的 print 命令，程序继续运行。通常，程序会接着被中断的命令之后继续运行，但对于 sleep 命令是个例外，如果在 sleep 命令运行时按下 Ctrl+C 组合键，程序将结束。从第 4 行以后，trap 将不会起作用。

3. 在脚本的主体部分，启动一个 while 循环。冒号是空命令，总是返回退出状态 0，因此将进入死循环。

4. 一旦进入循环，则调用函数 trapper。

5. 函数 trapper 中的 trap 命令在程序的这个部分不起作用，因为陷入是函数局部的。如果函数正常退出(即未按下 Ctrl+C 组合键)，脚本将继续执行。Ctrl+C 的默认行为是终止脚本的执行。

12.10 协作进程

协作进程是一个特殊的双向管道，允许 shell 脚本程序向另一条命令的标准输入写入数据，或是从它的标准输出读取数据。这为在当前程序中创建一个新接口提供了一种新的方法。把附加操作符 |& 置于命令尾部，就会将命令初始化成一个协作进程。普通的重定向



和后台处理不能应用在协作进程上。print 和 read 命令需要一个 -p 开关来从协作进程读和写。输出必须发送到标准输出，并且每条输出的消息必须以换行符结尾。每发送一条消息到标准输出，就必须刷新标准输出。可以通过使用带 >&p 或 <&p 操作符的 exec 命令来运行多个协作进程。例如，要打开文件描述符 4 作为一个协作进程，可以输入命令 exec 4>&p。

范例 12-65

(脚本)

```
    #!/bin/ksh
    # Scriptname: mycalculator
    # A simple calculator -- uses the bc command to perform the calculations
    # Because the shell performs operations on integers only, this program allows
    # you to use floating-point numbers by writing to and reading from the bcprogram.
1   cat << EOF
    **************************************************
2       WELCOME TO THE CALCULATOR PROGRAM
    **************************************************
3   EOF
4   bc |&                  # Open coprocess
5   while true
    do
6       print "Select the letter for one of the operators below "
7       cat <<- EOF
           a) +
           s) -
           m) *
           d) /
           e) ^
        EOF
8       read op
9       case $op in
           a) op="+";;
           s) op="-";;
           m) op="*";;
           d) op="/";;
           e) op="^";;
           *) print "Bad operator"
                 continue;;
        esac
10      print -p scale=3                      # write to the coprocess
11      print "Please enter two numbers: "    # write to standard out
12      read num1 num2                        # read from standard in
13      print -p "$num1" "$op" "$num2"        # write to the coprocess
14      read -p result                        # read from the coprocess
15      print $result
16      print -n "Continue (y/n)? "
17      read answer
18      case $answer in
        [Nn]* )
```

```
19          break;;
        esac
20  done
21  print Good-bye
```

(输出)

```
$ mycalculator
   ***********************************************
1      WELCOME TO THE CALCULATOR PROGRAM
   ***********************************************
6   Select one of the operators below
7       a) +
        s) -
        m) *
        d) /
        e) ^
   e
11  Please enter two numbers:
    2.3 4
    27.984
16  Continue (y/n)? y
6   Select one of the operators below
7       a) +
        s) -
        m) *
        d) /
        e) ^
   d
11  Please enter two numbers:
    2.1 4.6
    0.456
16  Continue (y/n)? y
6   Select one of the operators below
7       a) +
        s) -
        m) *
        d) /
        e) ^
   m
11  Please enter two numbers:
    4 5
    20
16  Continue (y/n)? n
21  Good-bye
```

说明

1. here 文档用于显示一个标题栏。
2. 打印这些文本，作为下面菜单的头部信息。
3. EOF 是一个用户定义的终止符，标志着文档输入结束。
4. 打开 dc 命令(桌面计算器)作为一个协作进程。它在后台执行。



5．启动 while 循环。由于 true 命令总是返回成功(退出状态为 0)，因此循环将无休止地运行，直至遇到 break 或 exit。

6．提示用户从所显示的菜单中选一项。

7．here 文档显示了一个数学运算符的列表，用户可以从中为 bc 程序选择运算符。

8．read 命令把用户的输入赋给变量 op。

9．case 命令从 op 中寻找匹配的值，并且赋一个操作符给 op。

10．带 -p 选项的 print 命令，将输出字符串“scale=3”通过管道传给协作进程 bc。bc 命令把 print 的输出接收为自己的输入，并且把 scale 赋值为 3(scale 定义了 bc 命令显示的数字中小数点后保留的位数)。

11．提示用户输入两个数字。

12．read 命令将用户输入赋给变量 num1 和 num2。

13．print -p 命令向 bc 协作进程发送算术表达式。

14．shell 从 bc 协作进程读取(read -p)，并且将读到的内容赋给变量 result。

15．显示计算的结果($result)。

16．询问用户是否继续操作。

17．用户进行输入，输入信息赋给变量 answer。

18．case 命令解析变量 answer。

19．如果用户输入 No、no 或 nope 等，则执行 break 命令，while 循环结束，控制转到程序 21 行。

20．关键字 done 标志着 while 循环结束。

21．循环结束时显示这一行。

12.11 调试

打开 noexec 选项或者使用 ksh 命令的 -n 参数，就可以不执行 ksh 脚本，而只是检查脚本的语法。如果脚本中有语法错，shell 就会报错。如何没有错，就不显示任何信息。

调试脚本的常用方法是打开 xtrace 选项或者使用 ksh 的 -x 命令。这些选项允许对脚本实施跟踪。在变量替换之后，脚本的每条命令就显示出来了，然后执行命令。当脚本的一行显示时，它前面会有一个 PS4 提示符，一个“+”符号。PS4 提示符的值可以修改。

当 verbose 选项打开或者通过 -v 选项调用 ksh(ksh -v scriptname)时，脚本中的每一行都将被显示，然后再执行。调试命令的使用参见表 12-14。

表 12-14 调试命令和选项

命　令	功　能	工作方式
export PS4='$LINENO'	PS4 提示符默认是一个“+”	改变提示符。这里是显示每行的行号
ksh -x scriptname	回显方式调用 ksh	变量替换之后，执行之前显示每一行脚本

(续表)

命　令	功　能	工 作 方 式
ksh - v scriptname	带 verbose 选项调用 ksh	执行之前显示每一行脚本，就好像是在命令行键入命令一样
ksh - n scriptname	带 noexec 选项调用 ksh	解释但不执每一行命令
set -x 或 set - o xtrace	打开 echo 选项	在脚本中进行跟踪
set +x	关闭 echo 选项	关闭跟踪
trap 'print $LINENO' DEBUG	打印脚本中每一行的 $LINENO 值	对于每个脚本命令，执行 trap 中的行为。参见 trap 的格式
trap 'print Bad input' ERR		如果返回非零的退出状态值，trap 就执行
typeset -ft	打开跟踪	在函数中跟踪执行过程
trap 'print Exiting from $0 'EXIT		当脚本或函数退出时显示信息

范例 12-66

(脚本)

```
    #!/bin/ksh
    # Scriptname: todebug
1   name="Joe Blow"
2   if [[ $name = [Jj]* ]] then
       print Hi $name
    fi
    num=1
3   while (( num < 5 ))
    do
4         (( num=num+1 ))
    done
5   print The grand total is $num
```

(输出)

```
1   $ ksh -x todebug
2   + name=Joe Blow
    + [[ Joe Blow = [Jj]* ]]
    + print Hi Joe Blow
    Hi Joe Blow
    + num=1        The + is the PS4 prompt
    + let num < 5
    + let num=num+1
    + let num < 5
    + let num=num+1
    + let num < 5
    + let num=num+1
    + let num < 5
    + let num=num+1
    + let num < 5
    + print The grand total is 5
5   The grand total is 5
```

说明

1. 带 - x 选项的 Korn shell 被调用。回显被打开，脚本每一行会显示在屏幕上，紧接



着是该行执行的结果。变量替换已经执行。-x 选项也可以用在脚本中，而不只用在命令行上，如：#!bin/ksh -x。

2. 行前带有加号“+”，即 PS4 提示符。
3. while 循环将被执行 4 次。
4. num 值每次递增 1。
5. while 循环结束后，打印结果。

范例 12-67

```
(脚本)
     #!/bin/ksh
     # Scriptname: todebug2
1    trap 'print "num=$num on line $LINENO"' DEBUG
     num=1
     while (( num < 5 ))
     do
         (( num=num+1 ))
     done
     print The grand total is $num
(输出)
     $ todebug2
2    num=1 on line 3
     num=1 on line 4
     num=2 on line 6
     num=2 on line 4
     num=3 on line 6
     num=3 on line 4
     num=4 on line 6
     num=4 on line 4
     num=5 on line 6
     num=5 on line 4
     The grand total is 5
     num=5 on line 8
     num=5 on line 8
```

说明

1. LINENO 是一个特殊的 Korn shell 变量，用来保存当前脚本行的行号。DEBUG 信号，与 trap 命令一起使用，使得脚本中每个命令执行时，括号中的字符串都被执行。
2. 当 while 循环执行时，将显示 num 变量的值和脚本行。

12.12 命令行

用 getopts 处理命令行选项

如果你编写的是一个包含多个命令行选项的脚本，那么使用位置参量并不是最有效的。例如，UNIX 的 ls 命令就有很多命令行选项和参数(一个选项需要一个前置的“–”号，而参数则不需要)。有几种途径可以将选项传递给程序：ls –laFi、ls –i –a –l –F、ls –ia –F 等。

如果有一个需要参数的脚本，位置参量可以用来单独处理这些参数，例如 ls –l –i –F。每个“–”选项会分别保存在变量$1、$2 和$3 中。但是，如果用户将所有的参数写在一起作为一个选项，例如 ls –liF，那会怎样呢？现在，–liF 将会赋给$1。getopts 函数可以像 ls 命令处理参数那样处理命令行参数[11]。getopts 函数允许 runit 程序使用各种组合来处理它的参数。

范例 12-68

（命令行）

```
1    $ runit -x -n 200 filex
2    $ runit -xn200 filex
3    $ runit -xy
4    $ runit -yx -n 30
5    $ runit -n250 -xy filey
```

（这些参数选项可以任意组合）

说明

1．runit 有 4 个参数：x 是一个选项，n 选项后需要紧跟一个数字，而 filex 则是一个单独的参数。

2．runit 将 x 选项和 n 选项的及数字参数 200 合在一起，filex 也是一个参数。

3．runit 合并了 x 和 y 选项。

4．runit 合并了 y 和 x 选项。－n 选项单独处理，其后是参数 30。

5．runit 合并了 n 选项和数字参数。x、y 选项被合并，filey 单独处理。

在讨论 runit 程序的详细内容之前，我们先来看如何用 getopts 来处理这些参数的，下面是 runit 中的一行命令：

```
while getopts :xyn: name
```

(1) x、y 和 n 是选项。

(2) 命令行选项以“-”或“+”号开头。

(3) 任何不包含“-”或“+”号的选项将向 getopts 函数表明该选项表结束了。

(4) 选项后的冒号表明需要一个参数，也就是说 -n 选项后需要一个参数。

(5) 选项前的冒号表明如果你输入了一个非法的选项，getopts 将允许程序员去处理它。例如，在命令 runit 中，-p 是一个不合法选项，getopts 将会告诉您这个问题，但 shell 不显示错误信息。

(6) 每次调用 getopts 时，它将在变量中置入下一个不带“-”号的选项(这里可以使用任何变量名)。如果选项前有 + 号，它则置入有 + 号的名字。如果给出了非法的参数，则名字将带一个 ? 号。如果缺少一个必须的参数，则用冒号取代该参数。

(7) OPTIND 是一个特殊的变量，初始化为 1，每次 getopts 处理完一个命令行参数后，OPTIND 递增为 getopts 要处理的下一个参数的序号。

(8) OPTARG 用来存放合法参数的值，如果给出了非法的选项，非法选项的值也保存在其中。

getopts 脚本范例 下面的范例脚本将演示 getopts 如何处理参数。

---

[11] 关于 C 库函数 getopts 的介绍可以参见 UNIX 或者 Linux 手册。



范例 12-69

（脚本）

```
    #!/bin/ksh
    # Program opts1
    # Using getopts -- First try --
1   while getopts xy options
    do
2   case $options in
3      x) print "you entered -x as an option";;
       y) print "you entered -y as an option";;
    esac
    done
--------------------------------------------------------------------------
```

（命令行）

```
4   $ opts1 -x
    you entered -x as an option
5   $ opts1 -xy
    you entered -x as an option
    you entered -y as an option
6   $ opts1 -y
    you entered -y as an option
7   $ opts1 -b
    opts1[3]: getopts: b bad option(s)
8   $ opts1 b
```

说明

1. getopts 用作 while 命令的条件。程序的有效选项列在 getopts 命令的后面，它们是 x 和 y。每个选项在循环体中被依次测试。每个选项将被赋给变量 option(不带前导的“-”号)。没有参数可处理时，getopts 将以非 0 状态退出，导致 white 循环结束。
2. case 命令用来测试选项变量中每个可能的选项：x 或 y。
3. 如果 x 是选项，则显示字符串“you entered x as an option”。
4. 命令行上，opts1 脚本的选项是 x 时，将会由 getopts 函数正常处理。
5. 命令行上，opts1 脚本的选项是 xy 时，将会由 getopts 函数正常处理。
6. 命令行上，opts1 脚本的选项是 y 时，将会由 getopts 函数正常处理。
7. 命令行上，opts1 脚本的选项是 b 时，getopts 将发出一条错误信息。
8. 不带“-”或“+”号的选项不能作为选项，这将导致 getopts 停止处理参数。

范例 12-70

（脚本）

```
    #!/bin/ksh
    # Program opts2
    # Using getopts -- Second try --
1   while getopts :xy options
    do
2       case $options in
        x) print "you entered -x as an option";;
        y) print "you entered -y as an option";;
```

```
3       \?) print $OPTARG is not a valid option 1>&2;;
        esac
    done
```

(命令行)

```
    $ opts2 -x
    you entered -x as an option
    $ opts2 -y
    you entered -y as an option
    $ opts2 xy
    $ opts2 -xy
    you entered -x as an option
    you entered -y as an option
4   $ opts2 -g
    g is not a valid option
5   $ opts2 -c
    c is not a valid option
```

说明

1. 选项表前的冒号是为了防止 Korn shell 在遇到错误选项时显示错误信息。如果选项错误，则将问号 ? 赋给 options 变量。

2. case 命令可用来测试问号，将错误信息打印到标准错误输出上。

3. 如果 options 变量被赋值为问号 ?，case 语句将会执行。问号前加上一个 "\"，其目的是防止 Korn shell 将它看作通配符而进行文件名替换。

4. g 不是合法选项。问号 ? 赋给了 options 变量，OPTARG 被赋值为非法选项 g。

5. c 不是合法选项，问号 ? 赋给了 options 变量，OPTARG 被赋值为非法选项 c。

范例 12-71

(脚本)

```
    #!/bin/ksh
    # Program opts3
    # Using getopts -- Third try --
1   while getopts :d options
    do
        case $options in
2       d) print -R "-d is the ON switch";;
3       +d) print -R "+d is the OFF switch";;
        \?) print $OPTARG is not a valid option;;
        esac
    done
 # Need the -R option with print or the shell tries to use -d as a print option
```

(命令行)

```
4   $ opts3 -d
    -d is the ON switch
5   $ opts3 +d
   +d is the OFF switch
```



```
6   $ opts3 -e
    e is not a valid option
7   $ opts3 e
```

说明

1. while 命令测试 getopts 的退出状态。如果 getopts 成功处理一个参数，则返回 0，并进入 while 循环体。选项前的冒号告诉 getopts，用户输入了无效的选项时不打印错误信息。

2. －d 是一个合法选项。如果输入－d 选项，则 d(不加－)将保存在 options 变量中。(print 命令的－R 选项允许 print 字符串中的第一个字符是“－”。)

3. +d 是一个合法选项，如果输入+d 选项，则 d(有+号)保存在 options 变量中。

4. -d 是 opts3 的一个合法选项。

5. +d 也是 opts3 的一个合法选项。

6. －e 选项无效。问号 ? 将保存到 options 变量中。非法的参数保存在变量 OPTARG 中。

7. 选项前无“－”也无“+”号。getopts 命令不去处去理它，返回非 0 值，white 循环结束。

范例 12-72

(脚本)

```
    #!/bin/ksh
    # Program opts4
    # Using getopts -- Fourth try --
1   alias USAGE='print "usage: opts4 [-x] filename " >&2'
2   while getopts :x: arguments
    do
    case $arguments in
3        x) print "$OPTARG is the name of the argument ";;
4        :) print "Please enter an argument after the -x option" >&2
            USAGE ;;
5       \?) print "$OPTARG is not a valid option." >&2
             USAGE;;
    esac
6   print "$OPTIND" # The number of the next argument to be processed
    done
-----------------------------------------------------------------------
```

(命令行)

```
7   $ opts4 -x
    Please enter an argument after the -x option
    usage: opts4 [-x] filename
    2
8   $ opts4 -x filex
   filex is the name of the argument
    3
9   $ opts4 -d
    d is not a valid option.
```

```
usage: opts4 [-x] filename
1
```

说明

1. 别名 USAGE 赋值为诊断信息，在 getopts 命令失败时，它会显示该信息。

2. while 命令测试 getopts 的退出状态。如果 getopts 可以成功地处理一个参数，则返回 0，进入 white 循环体。如果用户输入了无效的选项，选项前的冒号告诉 getopts 不出现错误信息。x 后面的冒号告诉 getopts，x 选项后必须跟一个参数。如果选项后跟一个参数，该参数将保存在 getopts 的内置变量 OPTARG 中。

3. 如果 x 选项带一个参数，该参数将存在 OPTARG 中，并将打印出来。

4. 如果没有给 x 提供参数，则在 arguments 变量中存放一个冒号，并显示适当的错误信息。

5. 如果输入了一个非法选项，则将问号 ？赋值给变量 arguments，并打印错误信息。

6. OPTIND 保存了接着要处理的选项的序号，其值总是比实际命令行参数的数目大 1。

7. x 选项需要一个参数，因而错误信息。

8. 参数名是 filex。变量 OPTARG 中存放的是 filex 参数的名称。

9. 选项 d 无效，打印出帮助信息。

12.13 安全性

12.13.1 特权脚本

当 Korn shell 用–p 选项调用一个脚本时，该脚本就成为特权脚本。当使用特权选项并且实际的 UID/GID 与有效的 UID/GID 不同时，文件.profile 不会被执行，而系统文件 /etc/suid_profile 将会被执行。

12.13.2 受限 shell

当 Korn shell 与–r 选项一起调用时，该 shell 就成为受限 shell。当 shell 受限时，不能使用 cd 命令，不能修改或重置 SHELL、ENV 和 PATH 变量。如果第一个字符是反斜杠，则命令不能执行。重定向操作符(>、<、|、>>)是非法的。该选项不能用 set 命令来设置或取消设置。可以用命令 rksh 来调用一个受限 shell。

12.14 内置命令

Korn shell 有很多内置命令，参见表 12-15。



表 12-15 内置命令及其功能

命 令	功 能
:	不做任何操作，返回 0
.file	"." 命令从文件 file 读取并执行命令
break	参见 12.6.6 节的 "break 命令"
continue	参见 12.6.6 节的 "continue 命令"
cd	改变目录
echo[args]	显示参数
eval command	shell 在执行命令前扫描命令行两次
exec command	代替当前 shell 执行 command 命令
exit [n]	退出 shell，返回状态值 n
export [var]	将 var 传递到子 shell
fc - e [editor] [lnr] first last	用来编辑历史列表中的命令。如果未指定编辑器，则使用 FCEDIT 中指定的编辑器。如果 FCEDIT 未设置，则调用默认编辑器/bin/ed。通常，命令 history 是指别名 fc - l 示例： fc - l　　列出命令历史列表中最近的 16 条命令 fc - e emacs grep　　读出最近的 grep 命令到 emacs 编辑器中 fc 25 30　　读取第 25 到第 30 条命令到 FCEDIT 指定的编辑器中默认为 ed 编辑器 fc - e -　　重新执行上次的命令 fc - e - Tom=Joe 28　　将历史命令中 28 号命令的 Tom 替换为 Joe fg　　将最近的后台作业移到前台 fg %n　　将作业号为 n 的作业移到前台。输入 jobs 以找到正确的作业号
jobs [- l]	按序号列出活动的作业，使用 - l 选项按 PID 来列出
	示例： **$ jobs** [3] + Runming　　sleep 50& [1] - Stopped　　vi [2] Running　　sleep%
kill [- signal process]	将信号发送到 PID 或作业号对应的进程。可参见/usr/include/sys/signal.h 中所列的信号
	信号： SIGHUP 1　　/*hangup (disconnect) */ SIGINT 2　　/*interrupt*/ SIGQUIT 3　　/* quit */ SIGILL 4　　/* illegal instruction (not reset when caught) */ SIGTRAP 5　　/* trace trap (not reset when caught) */

(续表)

命　令	功　能
kill [- signal process]	SIGIOT 6 /* IOT instruction */ SIGABRT 6 /* used by abort，replace SIGIOT in the future */ SIGEMT 7 /* EMT instruction */ SIGFPE 8 /* floating-point exception */ SIGKILL 9 /* kill (cannot be caught or ignored) */ SIGBUS 10 /* bus error */ SIGSEGV 11 /* segmentation violation */ SIGSYS 12 /* bad argument to system call */ SIGPIPE 13 /* write on a pipe with no one to read it */ SIGALRM 14 /* alarm clock */ SIGTERM 15 /* software termination signal from kill */ SIGURG 16 /* urgent condition on I/O channel */ SIGSTOP 17 /* sendable stop signal not from tty */ SIGTSTP 18 /* stop signal from tty */ SIGCONT 19 /* continue a stopped process */ 示例： 使用 kill 命令和信号名时，将 SIG 前缀去掉，并在信号名前加上短线 "-" kill —INT %3 kill —HUP 1256 kill —9 %3 kill %1
getopts	在 shell 脚本中分析命令行并检查合法选项
hash	列出所有追踪的别名
login [username]	用户登录
newgrp [arg]	将真实的组 ID 改为新的组 ID
print —[nrRsup]	替代 echo 命令。参见 prnit 命令
pwd	显示当前工作目录
read [var]	从标准输入读一行到变量 var 中
readonly [varr]	使变量只读，不可重置
return [n]	函数返回值为 n
set [-aefhknoptuvx- [-o option] [-A arrayname] [arg]]	
	示例： set　　列出所有变量和值 set +　　列出所有变量，但不显示值 set -o　　列出所有选项设置 set a b c　　重置位置参量$1、$2 和$3 set - s　　对$1、$2 和$3 按字母排序



(续表)

命　令	功　能
	set－o vi　设置 vi 选项
	set－xv　打开 xtrace 和 verbose 选项以用来调试
	set --　取消所有位置参量的设置
	set -- "$x"　将$1 设置为 x 的值
	set ＝＝%x　对 x 中的各项进行路径名扩展，然后设置各项的位置参量
	set -A nametom dick harry　name[0]设置为 tom name[1]设置为 dick name[2]设置为 harry
	set +A name joe n　ame[0]重置为 joe，数组的其余部分不变 name[1]是 dick name[2]是 harry
	使用－o 标志来设置选项。使用+o 标志来去除设置 set －o ignoreeof
	选项：
	allexport　设置后，导出所有已定义和修改的变量
	bgnice　以低优先级运行后台作业，而不是高优先级。它代替了 nice 选项
	emacs　设置 emacs 为内置编辑器
	errexit　当命令返回非 0 值时，shell 退出
	gmacs　设置 gmacs 为内置编辑器
	ignoreecf　忽略 EOF(即 Ctrl+D)键，防止用该键从 shell 终止，必须使用 exit 命令来退出 shell
	keyword　在命令行上，向 shell 环境中添加 keyword 参数
	markdirs　在所有的目录名上加上反斜扛以防止文件名扩展
	monitor　设置作业控制
	noclobber　防止使用重定向符>覆盖文件，使用>|来强制覆盖
	noexec　等同于 ksh－n。读取命令但不执行。目的是检查 shell 脚本的语法错误
	noglob　用 ksh 通配符元字符禁止文件名扩展
	nolog　函数定义不会存储在历史文件中
	nounset　如果变量未被设置则显示错误
	privileged　为 setuid 打开特权模式
	trackall　ksh 中每条命令都成为被跟踪的别名。对于交互式 shell 则自动打开
	verbose　回显每一行输入到屏幕上。调试时使用
	vi　设置 vi 为内置编辑器
	viraw　在某次输入时指定 vi 字符
	xtrace　扩展每条命令，并在 PS4 提示符中显示(变量已被扩展)
shift[n]	将位置参量左移 n 次

(续表)

命　令	功　能
times	显示从 shell 上启用进程的用户和系统时间
trap[arg][n]	当 shell 接受到信号(0、1、2 或 15)时，执行 arg
type[command]	打印命令类型，例如，pwd 是内置的 shell。在 ksh 中，则是 whence –v 的别名
typeset[options][var]	设置 shell 变量和函数的属性和值
ulimit[options size]	设置进程的最大限制
	示例： ulimit –a　显示所有限制值。 Time(seconds)unlimited. File(bloeks)unlimited. Data(kbytes)524280. Stack(kbytes)8192. Memory (kbytes) unlimited. Coredump (blocks) unlimited.
	其他选项： –c size　设置 core 转储值上限为 size 块 –d size　设置可执行的数据大小上限为 size 块 –f size　设置文件大小上限为 size 块 –m size　限制物理内存大小为 size KB –s size　限制栈空间大小为 size KB –t secs　限制进程执行时间为 secs 秒
umask [mask]	不带参数时，打印文件创建权限掩码
umask [octal digita]	用户文件创建模式掩码，针对用户、组及其他用户创建
unset [name]	清除变量或函数的设置
wait [pid#n]	等待 PID 为 n 的后台进程终止，并报告退出状态
whence [command]	打印命令的信息，如 ucb whereis
	示例： whence　–v happy　happy is a function whence　–v addon　addon is an unchefined funchtion whence　–v ls　ls is a tracked alias for /bin/ls whence　ls　/bin/ls

12.15 Korn shell 调用参数

当 ksh 调用时，可以使用参数控制其行为，见表 12-16。



表 12-16 ksh 参数

命令	功能
-a	自动导出所有变量
-c cmd	执行 cmd 命令字符串
-e	当一个命令返回非零值时退出
-f	关闭 globbing 功能，即文件名元字符扩展功能
-h	将命令作为可跟踪的别名
-i	设置为交互模式
-k	设置关键字选项，命令的所有关键参数将成为环境的一部分
-m	使命令在后台执行，以独立进程组方式运行，即使按下 Ctrl+C 组合键或用户登出，命令也继续运行，当作业运行完毕时，发送完成消息
-n	可用于调试。命令被扫描，但不被执行，可与 -x 和 -v 选项一起使用
-o	允许设置 12-15 表中 set 命令列出来的选项
-p	打开特权模式，用于运行 setuid 程序
-r	设置受限模式
-s	默认为从 stdin 读取命令
-t	在执行完 shell 输入的第一条命令且指定 -c 选项后，致使 shell 退出
-u	如果一条命令引用了未被设置的变量，将被当作是错误命令
-v	在任何命令分析、变量替换或其他处理之前，显示脚本的每一行或标准输入。输出将送到标准错误输出，以用于调试
-x	在执行前，显示脚本的每一行或标准输入，在输出上显示文件名扩展、变量替换和命令替换后的结果，所有输出将前置一个 PS4 提示符，即加号和一个空格。所有行将写到标准错误输出

习题 33 Korn shell 入门

1. 您使用的是什么 shell？您是如何知道的？

2. 在您的 HOME 目录中是否有.profile 和.kshrc？它们的区别是什么？什么是 ENV 文件？如何在修改后调用它？

3. 什么是默认主提示符？什么是默认次提示符？在命令行上改变主提示符以使它包含您的登录名？

4. 设置下列变量的目的是什么？

   a. set –o ignoreeof

   b. set –o noclobber

   c. set –o trackall

   d. ste –o monitor

   e. set –o vi

为什么要在 ENV 文件中设置这些变量？设置 PATH 的目的是什么？您的 PATH 变量都

有哪些元素？

5. 局部变量和环境变量的区别是什么？怎样列出所有变量？如何仅列出环境变量？列出当前的选项设置，输入：

```
set-o
```

哪些选项被打开？

6. 创建一个包含您全名的局部变量 myname 并导出该变量，键入：

```
ksh
```

变量导出了没有？键入 exit 回到父 shell，设置变量为只读，那么什么是只读变量？

7. 位置参量通常用作什么？

a. 键入：

```
set  apples  pears  peaches  plums
```

使用位置参量显示 plums，显示 apples peaches，显示 apples pears peaches plums，显示参数个数，重置位置参量为一个蔬菜的类，显示蔬菜的列表。fruit 表有变化？

b. 键入：

```
set --
print $*
```

输出有什么变化？

8. 显示当前 shell 的 PID，键入：

```
grep  $LOGNAME /etc/passwd
echo $?
```

$? 代表什么？退出状态值又向您表明所执行命令的什么结果？

9. 在.profile 文件中改变主提示符和次提示符，如何重新执行.profile 而不必注销后再次登录？

习题 34 命令行历史

1. 您的 HISTSIZE 变量设置为多少？HISTFILE 设成什么了？检查.kshrc 文件，看看是否存在 set - o vi，如果否，就在.kshrc 中设置，并重新执行该文件，键入：

```
. .kshre
```

2. 在命令行上键入下面的命令：

```
ls
date
who
cal 2 1993
date + %T
```

键入 history 或 fc - l，它们的功能是什么？反向打印这些命令，打印出无序号的历史表，打印当前和以前的 5 条命令。打印当前和第 10 条命令间的所有内容，打印 ls 命令和



cal 命令间的所有内容。

3．使用 r 命令，重新执行最近的命令，重新执行以字母 d 开头的最近的那些命令，改变 cal 命令的 year 为 1897，改变 date 命令的+%T 参数以显示当前时间。

4．如果设置了 history，在命令行上，按下 ESC 键，并且使用 K 键在历史表中上移，改变 ls 命令为 ls -alF 并且重新执行它。

5．键入 env 命令来检查 FCEDIT 变量是否被设置，如果没有被设置，在命令行上键入：

```
export FCEDIT=vi
```

键入：

```
fc -l -4
```

会输出什么结果？

6．在历史表中如何注释掉一行，以使该行在历史表中不被执行？

7．在命令行上键入：

```
touch a1 a2 a3 apples bears balloons a4 a45
```

现在使用表 10.2 和 10.3 中所示的历史 Esc 键序列，来完成如下操作：

a. 显示以 a 开头的第一个文件。

b. 显示以 a 开头的所有文件列表。

c. 显示以 b 开头的所有文件列表。

d. 显示命令并注释之。

8．在命令行上键入以下内容：

```
pnint a b c d e
```

9．使用历史 Esc unders coer 命令，将命令改为：

```
print e
```

使用历史 Esc under scoer 命令，将第一条命令改为：

```
print c
```

习题 35　别名与函数

1．用什么命令列出当前设置的所有别名？

2．用什么命令列出所有追踪别名？

3．为下列命令创建别名：

```
drecte +%T
history -n
ls -alF
rm -i
cp -i
print
```

4．怎样导出一个别名？

5. 创建一个包含以下命令的函数：

```
ls -F
print -n "The  time  is"
date +%T
print -n "Your  present  working  directory  is"
pwd
```

6. 执行该函数。
7. 现在创建自己的函数，且使用位置参量传递参数。
8. 用什么命令列出函数及其定义？
9. 试试 print 的一些选项。

习题 36 shell 元字符

1. 创建 meta 目录，且 cd 命令转向该目录，然后使用 touch 命令创建如下文件(touch 命令可以新建一个空文件或为已存在文件更新时间戳)：

```
abc  abc1  abc2  abc2191  Abc1  ab2  ab3  ab345  abc29  abc9  abc91
abc21xyz  abc2121  noone  nobody  nothing  nowhere
```

2. 执行以下操作。
   a. 列出文件名以小写字母 a 开头的所有文件。
   b. 列出文件名以 A 开头后跟两个字符的文件。
   c. 列出文件名以数字结尾的文件。
   d. 列出文件名为 abc 后跟一个数字的文件。
   e. 列出文件名匹配 nothing 或 noone 的文件。
   f. 列出文件名为 abc 后有一个或多个数字的文件。
   g. 列出文件名不包含模式 abc 的文件。
   h. 列出文件名为 ab 后紧跟 3 或 4 的文件。
   i. 列出文件名以 a 或 A 开头，后跟 b 且以数字结尾的文件。
   j. 当不匹配时出现什么错误消息？

习题 37 代字符号扩展、引号和命令替换

1. 使用代字符号执行以下任务。
   a. 显示主目录。
   b. 显示其他用户的主目录。
   c. 显示先前的工作目录。
   d. 显示当前的工作目录。
2. 什么变量保存的是当前工作目录？什么变量保存的是以前的工作目录？
3. 使用 - 转到先前的工作目录。
4. 使用 print 命令在屏幕输出以下内容(< >中的词是将要被扩展的变量名，[ ]中的词是已执行命令的输出。提示：使用命令替换)。

```
Hi <LOGNAME> how's your day going?
"No, <LOGNAME> you can't use the car tonight!", she cried.
```




```
The time is [ Sun Feb 22 13:19:27 PST 2004 ]
The name of this machine is [ eagle ] and the time is [ 31:19:27 ]
```

5. 创建包含用户名列表的文件，现创建一个包含用户名列表的变量 nlist，使用命令替换来引用它。

  a. 打印变量值，命令替换是怎样影响列表格式的？

  b. 将一个变量设置为 ps -eaf 的输出，并用它检验以上的操作。

  c. 其结果是什么？

习题 38 重定向

1. 打开编辑器，创建包含下面两行文本的文件 ex6。

```
Last time I went to the beach I found a sea shell.
While in Kansas I found a corn shell.
```

2. 现在将这一行添加到 ex6：The National Enquirer says someome gave birth to a shell, called the born shell。

3. 将 es6 用邮件发给你自己。

4. 使用管道，计算 ex6 文件的行数(wc - l)。

5. 列出所有设置，键入：

```
set -o
```

变量 noclobber 设置了没有？如果没有，键入：

```
set -o noclobber
```

结果怎样？

6. 键入：

```
cat << FINIS
How are you $LOGNAME
The time `date` Bye!!
FINIS
```

结果如何？

7. 使 tab 键入：

```
cat << -END
      hello there
      how are you
END
```

屏幕上显示什么？

8. 键入：

```
kat file2> error || print kat failed
```

结果如何？为什么？

9. 键入：

```
cat gombie2 > errorfile || print cat failed
```

结果如何？为什么？|| 操作符又是如何工作的，用您自己的命令去测试它。

10. 使用 find 命令，从根目录开始打印所有以 a 开头的文件，将标准输出定向到文件 foundit，并将错误定向到/dev/null。

习题 39 作业控制

1. 在命令行键入：

```
mail <user> 并按下 Ctrl+Z 组合键
```

再键入：

```
jobs
```

方括号中的数字是什么？

2. 键入：

```
sheep 300
jobs
bg
```

bg 做什么？+、 - 号表示什么？

3. 用作业控制终止 mail 作业。

4. 进入编辑器，键入^Z 停止作业。

现将已停止的 vi 作业移到后台，使用什么命令？

5. 键入以下命令：

```
jobs -l
```

6. 变量 TMOUT 用来做什么？

7. 内核花多少时间执行以下命令：

```
(sleep 5; ps-eaf )
```

习题 40 写一个名为 info 的 shell 脚本

1. 写一个名为 info 的脚本，在执行它之前，使用 chmod 命令将它的权限设置为可执行。
2. 为程序添加注释。
3. 程序执行时应做以下的事情：
   a. 输出登录的用户数。
   b. 输出时间和日期。
   c. 输出当前工作目录。
   d. 列出父目录中的所有文件。
   e. 显示所使用的 shell 名称。
   f. 显示 passwd 文件中包含登录名的一行。



g. 显示你的用户 ID。

h. 显示机器名。

i. 显示磁盘使用情况。

j. 显示本月的日历。

k. 向用户显示 good bye 并且显示非军事格式的时间。

习题 41　子串变量扩展

1. 写一个能执行以下的功能的脚本。

a. 将变量 mypath 设置成主目录。

b. 显示 mypath 的值。

c. 显示 mypath 中目录的最后一个元素。

d. 显示 mypath 中目录的第一个元素。

e. 显示 mypath 中除最后一个元素外的所有元素。

习题 42　lookup 脚本

1. 如果从本书合作站点下载的文件中没有提供文件 datafile，则创建文件 datafile，该文件由以下字段组成，由冒号隔开

a. 姓和名

b. 电话号码

c. 地址(街道、城市、州和区号)

d. 出生日期(04/12/66)

e. 工资

2. 在文件中添加 10 行代码，编写一个名为 lookup 的脚本，功能如下。

a. 致欢迎词。

b. 显示 datafile 中所有用户的名字和电话号码。

c. 显示 datafile 的行数。

d. 向用户显示 Good bye。

习题 43　使用 typeset

1. 写一个脚本，该脚本功能如下。

a. 让用户输入姓和名(英文名)。

b. 将输入结果存入两个变量中。

c. 使用新的 ksh 读命令。

2. 使用 typeset 命令转换第一个和最后一个名称变量转换为小写字母。

3. 检查人名，如果人名是 tom jones，则打印 Welcome，Tom jones。如果不是，则打印 Are you happy today，FIRSTNAME LASTNAME?(将用户的名字转换为大写字母)。

4. 让用户键入文本，给出问题的答案，并使用新的 ksh test 命令检查回答是 yes 还是 no。如果是 yes，让脚本打印出问候消息。如果是 no，则请用户休息一下并给出当前时间。

5. 重写 look up 脚本。

a. 脚本询问用户是否要向 datafire 中添加一行。

b. 如果用户回答 yes 或 y，则要输入：

名字

电话

地址

出生日期

工资

每一项都对应一个变量用于存放用户的输入。

示例：

```
print -n "what is the name of the person you are adding to the file?"
read name The information will be appended to the datafile.
```

习题 44 if/else 结构和 let 命令

1. 写一个脚本 grades，用于询问用户的测验成绩、分数：

a. 该脚本将检查分数的范围，即 0～100

b. 告诉用户得到的成绩是 A、B、C、D、E 还是 F

2. 写一个脚本 calc，以执行一个简单计算器的功能。脚本需提供一个简单的菜单：

```
[a] Add
[s] Subtract
[m] multiply
[d] Divide
[r] Remainder
```

3. 用户从菜单中选择其中的一个字母。

4. 提示用户输入 0~100 之间的两个整数。

5. 如果数字超出范围，打印错误信息并且退出脚本。

6. 程序对这两个数字进行计算。

7. 显示计算结果，分别以 10、8 或 16 为基数。

习题 45 case 语句

1. 写一个脚本 timegreet，功能如下。

a. 在脚本头部提供一个注释段，包括名字，日期和该程序的用途。

b. 将以下程序用 case 语句重写。

```
# The timegreet script by Ellie Quigley
you=$LOGNAME
hour=`date | awk '{print substr($4, 1, 2)}'`
print "The time is: $(date)"
if (( hour > 0 && $hour < 12 ))
then
    print "Good morning, $you!"
elif (( hour == 12 ))
then
    print "Lunch time!"
elif (( hour > 12 && $hour < 16 ))
```



```
then
    print "Good afternoon, $you!"
else
    print "Good night, $you!"
fi
```

2. 重写 lookup 脚本，用 if/elif 结构代替 case 命令，添加一个菜单：

```
1) Add Entry
2) Delete Entry
3) Update Entry
4) View Entry
5) Exit
```

习题 46 select 循环

1. 写一个脚本，功能是：

a. 脚本顶部提供一个注释段，包括名字，日期和本程序的用途。

b. 使用 select 循环做一个食品菜单，其输出类似于：

```
$ foods
1) steak and potatoes
2) fish and chips
3) soup and salad
Please make a selection. 1
Stick to your ribs
Watch your cholesterol
Enjoy your meal.
$ foods
1) steak and potatoes
2) fish and chips
3) soup and salad
Please make a selection. 2
British are coming
Enjoy your meal.
$ foods
1) steak and potatoes
2) fish and chips
3) soup and salad
Please make a selection. 3
Health foods...
Dieting is so boring.
Enjoy your meal.
$ foods
1) steak and potatoes
2) fish and chips
3) soup and salad
Please make a selection. 5
Not on the menu today!
```

2. 用 select 命令重写 look up 脚本，创建主菜单和子菜单，菜单如下。

```
1) Add Entry
2) Delete Entry
```

```
3) Update Entry
4) View Entry
    a) Name
    b) Phone
    c) Address
    d) Birthday
    e) Salary
5) Exit
```

习题 47 自动加载函数

函数自动加载的步骤：

(1) 创建目录 myfunctions。

(2) 转到 myfunctions 目录，用编辑器创建一个文件 good-bye。

(3) 在文件 good-bye 中插入函数 good-bye，与文件名相同。

(4) goodbye 函数。

```
function good-bye {
print The current time is $(date)
print "The name of this script is $0"
print See you later $1
print Your machine is `uname -n`
       }
```

(5) 写文件并退出编辑器，现在已创建了一个文件，其中函数名与文件名相同。

(6) 转到主目录，修改.kshrc 文件。

FPATH=$HOME/myfunctions

(7) 退出编辑器，用 dot 命令在当前环境中执行.kshrc 文件。

(8) 在 timegreet 脚本中添加以下两行：

```
autoload good-bye
good-bye $LOGNAME
```

(9) 运行 timegreet 脚本，good-bye 函数的输出将显示出来。

(10) 为 lookup 中的每个菜单项创建函数，将这些函数存在 myfunctions 目录下的 lookup_functions 文件中。

(11) 自动加载 lookup 脚本中的函数，并且针对不同情况执行相应的函数调用。

(12) 使用 trap 命令，让用户进入菜单选择，而不是直接输入整数值。出错时，trap 命令将打印错误信息，并由脚本请求用户重新输入类型正确的数据。



chapter

13



交互式 bash shell

13.1 简介

对于一个交互式的 shell 来说，标准输入、标准输出以及标准错误输出都是与终端相关联的。当以交互方式使用 Bourne Again shell(bash)时，在 bash 提示符下键入 UNIX/Linux 命令，然后等待它的响应。bash 给您提供了大量的各类内置命令和命令行快捷方式，例如历史、别名、文件和命令自动补全、命令行编辑等。标准 UNIX Bourne shell 和 Korn shell 已经具有了许多特性，而 GNU 项目又将 shell 扩展到包含许多新的特性，如增加了与 POSIX 的一致性。bash 2.x 版已经包含了 UNIX Korn shell 和 C shell 的许多特性，因此当 bash shell 与标准 Bourne shell 向上兼容时，无论在交互级还是在编程级都是一个全功能的 shell。对于 UNIX 用户来说，bash 提供了一个和标准 shell(sh、csh 和 ksh)的交互方式[①]。

本章主要讲述如何在命令行交互使用 bash 以及如何定制工作环境。我们将学习如何利用快捷方式和内置特性来创建一个高效有趣的工作环境。下一章将给您更深入的讲述。然后你就可以着手准备写 bash shell 脚本了，通过自动完成每天的工作和开发精细的脚本以更好地定制工作环境，如果您是一个管理者，那么这些工作的完成将不仅方便自己，也方便了整个用户组。

13.1.1 bash 版本

Bourne Again shell 是摩羯座的，它于 1988 年 1 月 10 日诞生，作者是 Brian Fox，后来由 Chet Ramey 对它进行维护、加强以及修改 bug。bash 的第一个版本是 0.99。现在的版本(到目前为止的)是 2.05 版，它主要是对 2.0 版本进行了加强，但是仍有许多操作系统使用 1.14.7 版。所有的版本都可以在 GNU 的公共许可下自由获取[②]。为了知道你所使用的是哪个版本，可以使用 bash 的--version 选项或者打印环境变量 BASH_VERSION 的值。

① 虽然传统上 bash 是 Linux 平台的默认 shell，但现在它已经和 Solaris 8 绑定。

② 可以通过访问 www.delorie.com/gnu/来获得 bash 的最新版本。



范例 13-1

```
(UNIX)
$ bash --version
GNU bash, version 2.05.0(1)-release (sparc-sun-solaris)
Copyright 2000 Free Software Foundation, Inc.

$ echo $BASH_VERSION
2.05.0(1)-release
```

范例 13-2

```
(Linux)
$ bash --version
GNU bash, version 2.05.0(1)-release (i386-redhat-linux-gnu)
Copyright 2000 Free Software Foundation, Inc.
```

13.1.2 启动

如果登录 shell 是 bash shell，那么在显示提示符之前，会先运行一组进程。

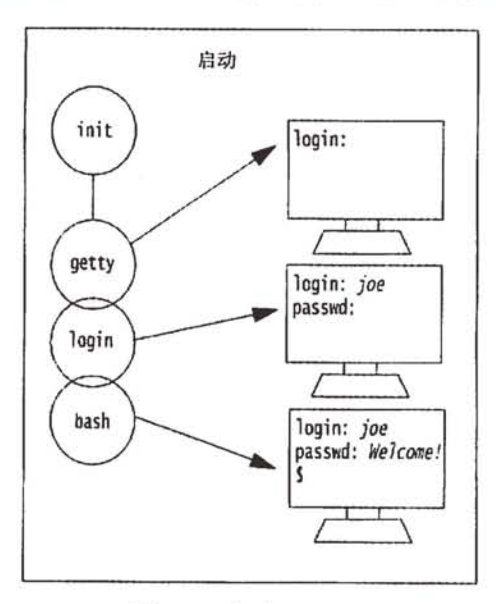


图 13-1 启动 bash shell

系统启动后运行的第一个进程是 init，它的进程标识符(PID)是 1。init 派生一个 getty 进程。该进程负责打开终端端口，提供标准输入的来源，以及标准输出与标准错误输出的去处，并且在屏幕上显示一个登录提示符。接下来执行的是/bin/login 程序。login 程序依次执行下面这些工作：提示用户输入口令、加密并验证用户输入的口令、设置初始环境、启动用户的登录 shell(登录 shell 是 passwd 文件的最后一项，对本章而言，就是/bin/bash)。bash 进程首先查找系统文件/etc/profile，并且执行其中的命令。然后它在用户的主目录中查找一个名为.bash_profile 的初始化文件[3]。执行完.bash_profile 中的命令后，bash shell 接着在用

③ bash 使用许多不同的初始化文件，它们将在下一小节讨论。

户的 ENV 文件，通常叫做.bashrc，中执行一个命令。最后默认的美元($)提示符将出现在屏幕上，shell 开始等待用户输入命令(有关初始化文件的更多内容，请参见 13.2 节，“环境”)。

在命令行更改 shell 如果想从命令行临时启动另外一个 shell(不改变/etc/passwd 文件)，只需键入 shell 名就可以了。例如，当前使用的是标准的 Bourne shell，而您想用 bash 作为你的 shell，只需在命令行简单地键入 bash 就可以改变 shell。

范例 13-3

```
1  $ ps
   PID   TTY    TIME   CMD
   1574  pts/6  0:00   sh
2  $ bash
   bash-2.05$
3  bash-2.05$ ps
   PID   TTY     TIME   CMD
   1574  pts/6   0:00   sh
   1576  pts/6   0:00   bash
```

说明

1. ps 命令的输出显示正在运行的进程。当前，sh(Bourne shell)正在运行。

2. 在 Bourne shell 提示符下，用户键入 bash 启动 Bourne Again shell。出现了一个新的提示符。

3. 在 bash 提示符下，执行 ps 命令。输出显示两个 shell 在运行，而且当前的 shell 是 bash。

13.2 环境

一个进程的环境包括：变量、打开的文件、当前的工作目录、函数、资源限额、信号等。它定义了可以从一个进程继承到下一个进程的特性，以及对当前工作环境的配置。用户 shell 的配置定义在 shell 初始化文件中。

13.2.1 初始化文件

Bash shell 有许多启动文件，这些文件是可以执行 source 命令的。对一个文件执行 source 命令会使这个文件中的所有设置成为当前 shell 的一部分。也就是说，不会创建子 shell(source 命令在 13.2.6 一节，“source 或 dot 命令”中讨论)。初始化文件是否执行 source 命令取决于 shell 是一个登录 shell，一个交互式 shell(但不是登录 shell)还是一个非交互式 shell(一个 shell 脚本)。

登录时，如果在用户的主目录下存在.bash_profile 文件，就对其执行 source 命令。它先设置用户的别名和函数，再设置用户特定的环境变量以及启动脚本。

如果用户没有.bash_profile 文件，但有一个名为.bash_login 的文件，那么将对这个文件



执行 source 命令，如果也没有.bash_login 文件，而有一个.profile 文件，就对这个.profile 文件执行 source 命令。

/etc/profile 文件　/etc/profile 文件是一个系统级的初始化文件，由系统管理员进行设置，在用户登录时执行指定的任务。这个文件在 bash shell 启动时被执行。它可以被系统上的所有 Bourne shell 和 Korn shell 用户使用，通常执行诸如在邮件假脱机程序中查找新邮件、显示文件/etc/motd 中的当日信息之类的任务(在学完本章之后，您会对下面这个范例有更深刻的理解)。

范例 13-4

```
(/etc/profile 示例)
# /etc/profile
# Systemwide environment and startup programs
# Functions and aliases go in /etc/bashrc
1  PATH="$PATH:/usr/X11R6/bin"
2  PS1="[\u@\h \W]\\$ "
3  ulimit -c 1000000
4  if [ `id -gn` = `id -un` -a `id -u` -gt 14 ]; then
5     umask 002
   else
      umask 022
   fi
6  USER=`id -un`
7  LOGNAME=$USER
8  MAIL="/var/spool/mail/$USER"
9  HOSTNAME=`/bin/hostname`
10 HISTSIZE=1000
11 HISTFILESIZE=1000
12 export PATH PS1 HOSTNAME HISTSIZE HISTFILESIZE USER LOGNAME MAIL
13 for i in /etc/profile.d/*.sh ; do
14    if [ -x $i ]; then
15          . $i
      fi
16 done
17 unset i #
```

说明

1. 给 PATH 变量赋值，shell 将其指定的位置搜索命令。

2. 指定主提示符。它将用户名(\u)、@符号、主机(\w)以及一个美元符显示在 shell 窗口。

3. 设置 ulimit 命令(shell 内置命令)来限制所创建的 core 文件的最大容量为 1000000 字节。core 文件是已经破碎的程序的信息转储器，它们占用许多磁盘空间。

4. 这一行可解释为：如果用户的组名等于用户名，并且用户的 ID 号大于 14 则继续执行第 5 行。

5. 将 umask 设为 002。创建的目录权限为 775，文件权限为 664。否则，umask 设为 022，目录权限为 755，文件权限为 644。

6. 将用户名(id　-un)赋值给变量 USER。

7. 将$USER 中的值赋给变量 LOGNAME。

8. 将保存用户邮件的邮件假脱机文件的路径赋给变量 MAIL。

9. 给 HOSTNAME 变量赋值用户的主机名。

10. 将 HISTSIZE 变量设为 1000。HISTSIZE 控制所记下并在 shell 退出后保存到历史文件中的历史项(储存在 shell 内存的历史清单中)的个数。

11. 设置 HISTFILESIZE 以限制存储在历史文件中的命令数为 1000，即，历史文件中 1000 行以后内容的将被删除(请参见 11.5.2，“历史”)。

12. 输出出这些变量，这样子 shell 和子进程也可以识别它们。

13. 循环结构，遍历 etc/profile.d 目录下的每个文件(以.sh 结尾)，执行 done 之前的命令。

14. 检查文件是否是可执行文件，如果是，则执行下一行命令。

15. 用点命令执行(即 source)这个文件。在/etc/profile.d 目录下的 lang.sh 和 mc.sh 文件，一个进行字符和字体设置，另一个创建一个名为 mc 的函数，该函数启动一个名为 Midnight Commander 的 visual/browser 文件管理程序。要知道文件管理器如何工作，可以在 bash 提示符下键入 mc。

16. done 关键字标志着 for 循环的结束。

17. 变量 i 被复位，也就是说，从 shell 的名字空间里删除。如果不再给 i 赋值，那么在 for 循环里给它赋了什么值，它就是什么值。

~/.bash_profile 文件 如果在用户的主目录下找到~/.bash_profile 文件，那它将在/etc/profile 文件后被执行 source 命令。如果~/.bash_profile 文件不存在，bash 将寻找另外一个用户定义的文件~/.bash_login，然后对它执行 source 命令，如果~/.bash_login 也不存在，bash 将对~/.profile(如果它存在)文件执行 source 命令。只能对这三个文件(~/.bash_profile、~/.bash_login 或~/.profile)中的一个执行 source 命令。bash 还将检查用户是否有一个.bashrc 文件并对它执行 source 命令。

范例 13-5

```
(.bash_profile 示例)
   # .bash_profile
   # The file is sourced by bash only when the user logs on.
   # Get the aliases and functions
1  if [ -f ~/.bashrc ]; then
2      . ~/.bashrc
   fi
   # User-specific environment and startup programs
3  PATH=$PATH:$HOME/bin
4  ENV=$HOME/.bashrc     # or BASH_ENV=$HOME/.bashrc
5  USERNAME="root"
6  export USERNAME ENV PATH
7  mesg n
8  if [ $TERM = linux ]
   then
      startx    # Start the X Window system
   fi
```



说明

1. 如果在用户的主目录下有一个名为.bashrc 的文件则继续执行第 2 行。

2. 为登录 shell 运行(source).bashrc 文件。

3. 给 PATH 变量添加用户 bin 目录的路径，通常 shell 脚本保存在该位置。

4.将 BASH_ENV[4](ENV)文件设成.bashrc 文件的路径名，只有设置了 BASH_ENV(ENV)变量，才能为交互式 bash shell 和脚本对.bashrc 初始化文件执行 source 命令。.bashrc 文件包含用户定义的别名和函数。

5. 将变量 USERNAME 设为 root。

6. 将这些变量输出，这样子 shell 和其他进程都可以识别它们。

7. 带 n 选项执行 mesg 命令，将禁止其他命令写入终端。

8. 如果 TERM 变量的值是 Linux，那么 startx 将启动 X Windows 系统(允许多个虚拟控制台的图形用户接口)，而不是在 Linux 控制台窗口启动一个交互式会话。因为只有在登录时才对~/.bash_profile 文件执行 source 命令，所以登录 shell 是启动 X Windows 会话最好的地方。

BASH_ENV(ENV)变量 从 BASH 2.0 版开始，BASH_ENV 文件简称为 ENV 文件(和 Korn shell 一样)。BASH_ENV(ENV)变量在~/.bash_profile 文件中设置。将每次交互式 bash shell 或 bash 脚本启动时要执行的文件名赋值给该变量。BASH_ENV(ENV)文件包含特定的 bash 变量和别名。通常命名为.bashrc，也可以是其他名称。当特权选项打开(bash –p 或设置-o 特权)或使用--norc 命令行选项(bash –norc 或 bash --norc)时，将不处理 BASH_ENV(ENV)文件。

.bashrc 文件 BASH_ENV(ENV)变量被赋值(按惯例)为名称.bashrc。每次当一个新的或交互式 bash shell 或 bash 脚本启动时自动对这个文件执行 source 命令。它包含那些只属于 bash shell 的设置。

范例 13-6

```
(.bashrc 示例)
# If the .bashrc file exists, it is in the user's home directory.
# It contains aliases (nicknames for commands) and user-defined function
# .bashrc
# User-specific aliases and functions
1   set -o vi
2   set -o noclobber
3   set -o ignoreeof
4   alias rm='rm -i'
    alias cp='cp -i'
    alias mv='mv -i'
5   stty erase ^h
    # Source global definitions
6   if [ -f /etc/bashrc ]; then
        . /etc/bashrc
    fi
```

④ bash 2.0 以后可以使用 BASH_ENV。

```
7   case "$-" in
8       *i*) echo This is an interactive bash shell
             ;;
9       *)   echo This shell is noninteractive
             ;;
    esac
10  history_control=ignoredups
11  function cd { builtin cd $1; echo $PWD; }
```

说明

1. 带-o 开关的 set 命令将打开或关闭特定的内置选项(请参见 13.2.2，“set-o 选项”)。如果开关是-o、一个减号，选项被打开，如果是一个加号，选项被关闭。vi 选项允许编辑交互式命令行，例如，设置成-o vi 将打开交互式命令行编辑，反之设置成 vi +o 则关闭它(请参见表 13-1)。

表 13-1　内置 set 命令选项

选 项 名	快捷开关	含　义
allexport	-a	从这个选项被设置开始就自动标明要输出的新变量或修改过的变量，直至选项被复位
braceexpand	-B	打开花括号扩展，它是一个默认设置
emacs		使用 emacs 内置编辑器进行命令行编辑，是一个默认设置
errexit	-e	当命令返回一个非零退出状态(失败)时退出。读取初始化文件时不设置
histexpand	-H	执行历史替换时打开!和!!扩展，是一个默认设置
history		打开命令行历史、默认为打开
ignoreeof		禁止用 EOF(Crtl+D)键退出 shell。必须键入 exit 才能退出。等价于设置 shell 变量 IGNOREEOF=10
keyword	-k	将关键字参数放到命令的环境中
interactive-comments		对于交互式 shell，把#符后面的文本作为注释
monitor	-m	设置作业控制
noclobber	-C	防止文件在重定向时被重写
noexec	-n	读命令，但不执行。用来检查脚本的语法。交互式运行时不开启
noglob	-d	禁止用路径名扩展。即关闭通配符
notify	-b	后台作业完成时通知用户
nounset	-u	扩展一个未设置的变量时显示一个错误信息
onecmd	-t	在读取和执行命令后退出
physical	-P	设置时，在键入 cd 或 pwd 时禁止符号链接。用物理目录替代
posix		如果默认操作不符合 POSIX 标准就改变 shell 的行为



(续表)

选 项 名	快捷开关	含 义
privileged	-p	设置后，shell 不读取.profile 或 ENV 文件，且不从环境继承 shell 函数。将自动为 setuid 脚本开启特权
verbose	-v	为调试打开 verbose 模式
vi		使用 vi 内置编辑器进行命令行编辑
xtrace	-x	为调试打开 echo 模式

2. 打开 noclobber 选项，它使得用户在使用重定向时不能重写文件。如 sort filex > filex。(参见 13.17，“标准 I/O 和重定向”)。

3. 退出 shell 时，通常可键入^D。如果设置了 ignoreeof，必须键入 exit 才能退出。

4. rm 的别名，rm -i，使得 rm 成为交互式的，因而它会在真正删除文件前询问用户以进行确定。cp 的别名，cp -i，使得复制也成为交互式的。

5. stty 命令用来将终端退格键设置为删除键。^H 代表退格键。

6. 如果/etc/bashrc 文件存在，就对它执行 source 命令。

7. 如果 shell 是交互式的，那么专用的变量$-将包含一个 i 字符。如果不是，可能是正在运行一个脚本。case 命令对$-求值。

8. 如果从$-返回的值匹配*i*(也就是说，包含 i 的任意串)，那么 shell 将打印出“This is an interactive bash shell”。

9. 否则，shell 打印“This shell is noninteractive”。在提示符下启动一个脚本，或一个新的 shell 时，将被告知 shell 是否是交互式的。由此我们可以理解“交互式”和“非交互式”的含义。

10. history_control 设置用来控制在历史文件中保存多少条命令。历史文件中已有的命令将不再保存，即忽略重复。

11. 这是一个用户定义的函数。当用户改变目录时，将打印当前的工作目录 PWD。这个函数被命名为 cd，且包含它的定义——cd 命令。函数定义中专用的内置命令 builtin 在 cd 之前执行，以防函数进入一个死循环。即防止它无限地调用自己。

/etc/bashrc 文件 系统级的函数和别名可以在/etc/bashrc 文件中设置。主提示符也常在这里设置。

范例 13-7

```
(/etc/bashrc 示例)

# Systemwide functions and aliases
# Environment stuff goes in /etc/profile

# For some unknown reason bash refuses to inherit
# PS1 in some circumstances that I can't figure out.
# Putting PS1 here ensures that it gets loaded every time.
```

```
1   PS1="[\u@\h \W]\\$ "
2   alias which="type -path"
```

说明

1. 系统级的函数和别名在这里设置。bash 的主提示符设为用户名(\u)、@符号、主机名(\h)，当前工作目录的基本名和一个美元符(请参见表 13-2)。这个提示符将出现在所有的交互式 shell 中。

表 13-2　提示符串设置

反斜杠序列	含　义
\d	日期是“星期　月　日”的格式(如，Tue May 26)
\h	主机名
\n	换行符
\nnn	对应于八进制数 nnn 的字符
\s	shell 的名称，$0 的基名(最后一个斜线后面的部分)
\t	当前时间是 HH:MM:SS 格式
\u	当前用户的用户名
\w	当前的工作目录
\W	当前工作目录的基本名
\#	该命令的编号
\!	该命令的历史编号
\$	如果有效的 UID 是 0，是一个#号，否则是$
\\	反斜杠
\[	开始一个非打印字符序列。可以用来在提示符中嵌入一个终端控制序列
\]	非打印字符序列的结尾
bash 2.x 版以上新增：	
\a	ASCII 报警字符
\e	ASCII 擦除符(033)
\H	主机名
\T	以 12 小时格式表示的当前时间：HH:MM:SS
\v	bash 的版本，如，2.03
\V	bash 的发行号和路径级别，如，2.03.0
\@	以 12 小时制 AM/PM 格式表示的当前时间

2. 通常在用户的.bashrc 文件中设置别名，即命令的简称。当 bash 启动时，别名被预置，且可用。当想找出一个程序在磁盘的什么位置(即在哪个目录下可以找到它)时，可以使用该命令。例如，which ls 将打印出/bin/ls。



~/.profile 文件 profile 文件是一个用户定义的初始化文件。它在用户的主目录下，只有当运行 sh(Bourne shell)登录时才对它执行 source 命令。由于此文件被 Bourne shell 使用，因此不能包含任何针对 bash 的特定设置。如果运行 bash，那只有在没有找到上面列出的任何初始化文件时才运行.profile 文件。它允许用户定制和修改 shell 环境。环境和终端设置通常放在这里，如果一个窗口应用程序或数据库应用程序需要初始化，也在这里启动。

范例 13-8

```
(.profile 示例)
# A login initialization file sourced when running as sh or the
# .bash_profile or
# .bash_login are not found.

1   TERM=xterm
2   HOSTNAME=`uname -n`
3   EDITOR=/bin/vi
4   PATH=/bin:/usr/ucb:/usr/bin:/usr/local:/etc:/bin:/usr/bin:.
5   PS1="$HOSTNAME $ > "
6   export TERM HOSTNAME EDITOR PATH PS1
7   stty erase ^h
8   go () { cd $1; PS1=`pwd`; PS1=`basename $PS1`; }
9   trap '$HOME/.logout' EXIT
10  clear
```

说明

1. TERM 变量被赋值为终端的类型，xterm。

2. 因为 uname -n 命令被括在反引号内，所以 shell 将执行命令替换，即，将命令的输出(主机的名字)赋给变量 HOSTNAME。

3. 变量 EDITOR 被设为/bin/vi。mail 和 history 之类的程序将在设置编辑器时用到这个变量。

4. 变量 PATH 的值被设为一组目录项，shell 查找 UNIX 程序时要搜索这些目录。例如，如果键入 ls 命令，shell 就会查找 PATH 列出的目录，直到它在其中某个目录下找到 ls 程序为止。如果未能找到指定的程序，shell 会告诉您这个结果。

5. 把主提示符设置为 HOSTNAME 的值(即机器名)和符号$与>。

6. 输出所列的全部变量。由这个 shell 启动的所有子程序都能识别它们。

7. stty 命令用于设置终端选项，擦除键被设置为^H，这样，当您按下退格键时，光标前面那个字符就会被删除。

8. 定义一个名为 go 的函数。这个函数的目的是接收一个目录名参数，使用 cd 命令进入该目录，并将主提示符设置为当前工作目录。basename 命令删除路径的各分量，只保留最后一个。这样，提示符就可以显示当前的目录。

9. trap 命令是一个信号处理命令。退出 shell(即注销时)，shell 会执行.logout 文件。.logout 文件是一个用户定义文件，它包含那些在注销前执行的命令，这些命令将完成清除临时文件，记录注销时间等任务。

10. clear 命令清空屏幕。

~/.bash-logout 文件 用户注销(退出登录 shell)时，如果~/.bash_logout 文件存在，就执行 source 命令。该文件包含一些命令，这些命令将完成清除临时文件、清空历史文件、记录管理信息等任务。

禁止执行启动文件的选项 如果带--noprofile 选项运行 bash(如，bash --noprofile)，那么将不会对启动文件/etc/profile、~/.bash_profile、~/.bash_login 或~/.profile 执行 source 命令。

如果用-p 选项调用 bash(如 bash –p)，那么 bash 将不会读取用户的~/.profile 文件。

如果 bash 被当作 sh(Bourne shell)调用，它将尽可能像地模仿 Bourne shell 的行为。对一个登录 shell 来说，它只试图对/etc/profile 和~/.profile 执行 source 命令。--noprofile 选项仍可以用来禁止这种行为。如果 shell 被当作 sh 调用，它不会尝试对任何其他的启动文件执行 source 命令。

.inputrc 文件 另一个默认的初始化文件.inputrc，也会在 bash 启动时被读取。这个文件(如果在用户的主目录下存在的话)，包含定制键击行为的变量和将串、宏、控制函数和键绑定的设置。绑定键的名字和它们做什么可以在 readline 库中找到，这个库由处理文本的应用程序使用。绑定键主要在执行命令行编辑时，被内置的 emacs 和 vi 编辑器使用(有关 readline 的更多内容请参见 13.5.4 一节，“命令行编辑”)。

13.2.2 用内置的 set 和 shopt 命令设置 bash 选项

set –o 选项 当使用开关-o 时，set 命令可以设置选项。选项可以用来定制 shell 环境。它们不是打开就是关闭，通常在 BASH_ENV(ENV)文件中设置。set 命令的许多选项都有一个简写格式。例如，set –o noclobber 可以写成 set –C(请参见前面的表 13-1)。

格式

```
set -o option # Turns on the option.
set +o option # Turns off the option.
set -[a-z]    # Abbreviation for an option; the minus turns it on.
set +[a-z]    # Abbreviation for an option; the plus turns it off.
```

范例 13-9

```
1  set -o allexport
2  set +o allexport
3  set -a
4  set +a
```

说明

1. 设置 allexport 选项。这个选项使所有的环境变量自动导出到子 shell 中。
2. 复位 allexport 选项。现在所有的环境变量都将成为当前 shell 的局部变量。
3. 设置 allexport 选项。同 1。但不是每个选项都有一个简写(请参见表 13-1)。
4. 复位 allexport 选项。同 2.

范例 13-10

```
1 $ set -o
  braceexpand          on
  errexit              off
```




```
  hashall              on
  histexpand           on
  keyword              off
  monitor              on
  noclobber            off
  noexec               off
  noglob               off
  notify               off
  nounset              off
  onecmd               off
  physical             off
  privileged           off
  verbose              off
  xtrace               off
  history              on
  ignoreeof            off
  interactive-comments on
  posix                off
  emacs                off
  vi                   on
2  $ set -o noclobber
3  $ date > outfile
4  $ ls > outfile
   bash: outfile: Cannot clobber existing file.
5  $ set +o noclobber
6  $ ls > outfile
7  $ set -C
```

说明

1. 用-o 选项，set 命令列出所有当前设置的和复位的选项。

2. 用-o 设置选项。noclobber 选项被设置。它禁止在重定向时重写文件。没有设置 noclobber 时，>号后面的文件若存在，就会被覆盖，不存在时则会被创建。

3. UNIX/Linux date 命令的输出被重定向到文件 outfile。

4. 这次，outfile 文件已存在。当试图再次将 ls 的输出重定向到 outfile 时，shell 会告知文件已存在。如果 noclobber 选项没有设置，文件将被覆盖。

5. 使用带+o 选项的 set 命令关闭 noclobber 选项。

6. 这次，覆盖 outfile 文件成功，因为没有设置 noclobber。

7. 带-C 开关的 set 命令是开启 noclobber 的另一种法，+C 选项将关闭它。

内置 shopt(2.x 以上版) 在更新版本的 bash 中，shopt(shell 选项)内置命令是 set 命令的一种替代。shopt 在许多方面和 set 内置命令一样，但它为配置 shell 增加了更多的选项。13.19.2 节中的表 13-27 列出了所有的 shopt 选项。在下面的例子中，带-p 选项的 shopt 打印所有可用的选项设置。-u 开关表示一个复位的选项，-s 表示选项当前被设置。

范例 13-11

```
1  $ shopt -p
   shopt -u cdable_vars
```

```
    shopt -u cdspell
    shopt -u checkhash
    shopt -u checkwinsize
    shopt -s cmdhist
    shopt -u dotglob
    shopt -u execfail
    shopt -s expand_aliases
    shopt -u extglob
    shopt -u histreedit
    shopt -u histappend
    shopt -u histverify
    shopt -s hostcomplete
    shopt -u huponexit
    shopt -s interactive_comments
    shopt -u lithist
    shopt -u mailwarn
    shopt -u nocaseglob
    shopt -u nullglob
    shopt -s promptvars
    shopt -u restricted_shell
    shopt -u shift_verbose
    shopt -s sourcepath
2   $ shopt -s cdspell
3   $ shopt -p cdspell
    shopt -s cdspell
4   $ cd /hame
    /home
5   $ pwd
    /home
6   $ cd /ur/lcal/ban
    /usr/local/man
7   $ shopt -u cdspell
8   $ shopt -p cdspell
    shopt -u cdspell
```

说明

1. 带-p(打印)选项，shopt 命令列出所有可设置的 shell 选项和它们的当前值——设置(-s)或复位(-u)。

2. 带-s 选项，shopt 设置(打开)一个选项。cdspell 选项使 shell 能小范围地纠正作为 cd 命令参数的目录名的拼写错误。它能纠正简单的键入错误、插入遗漏的字母、甚至调换字母的顺序。

3. 带-p 选项和一个选项名，shopt 将显示该选项是否被设置。结果显示选项被设置(-s)。

4. 在这个例子中，用户试图转向他的主目录，但是将 home 拼错了。于是 shell 会对它进行修改。即，将 hame 改成 home。并改变目录为/home。

5. pwd 命令的输出显示的是当前工作目录，可以看出目录确实改变了，甚至在用户拼写错误的情况下。

6. 这次目录名缺少一个字母且最后一项 ban 拼错了。shell 通过插入遗漏的字母并将 ban



改正为 man 来试着拼出正确的路径名。因为 ban 中拼错的 b 是第一个字符，shell 将在目录中搜索一个可能以 a 和 n 结尾的项。它找到了 man。

7. 用-u 开关[⑤]，shopt 复位(或关闭)选项。

8. 用-p 开关和选项名，shopt 显示该选项是否被设置。结果显示 cdspell 选项已经被复位(-u)。

13.2.3 提示符

交互式使用时，shell 会提示用户进行输入。看到提示符，就可以开始输入命令了。bash shell 提供 4 种提示符：主提示符是美元符号($)，次提示符则是一个向右的尖括号(>)。保存第 3 和第 4 提示符的变量分别是 PS3 和 PS4，将在以后讨论。交互运行时，shell 会显示这两个提示符。注意，我们可以改变提示符的默认值。

变量 PS1 中保存的是主提示符。登录并等待用户输入(通常是一个 UNIX/Linux 命令)时，它的值——美元符号将出现。变量 PS2 则保存次提示符，其初值是向右的尖括号(>)。如果用户在将命令输完整之前按了回车键，屏幕上就会显示次提示符。改变主提示符和次提示符的命令将在后面给出。

主提示符　默认的主提示符是美元符(或 bash $)。您可以改变自己的主提示符。提示符通常在/etc/bashrc 文件或用户的初始化文件.bash_profile 或.profile(Bourne shell)中定义。

范例 13-12

```
1  $ PS1="$(uname -n) > "
2  chargers >
```

说明

1. 默认的主提示符是美元符(bash $)。这条命令把提示符 PS1 重置为机器名[⑥] (uname -n)和符号>。

2. 显示新的提示符。

用专用转义序列设置提示符　可以通过在提示符串中插入专用的反斜杠/转义序列来定制提示符。13.2.1 节表 13-2 列出了专用序列。

范例 13-13

```
1  $ PS1="[\u@\h \W]\\$ "
   [ellie@homebound ellie]$
2  $ PS1="\W:\d> "
   ellie:Tue May 18>
```

说明

1. 用专用的反斜杠/转义序列定制主 bash 提示符。\u 代表的是用户的登录名，\h 是主机名，\W 是当前工作目录的基名。有两个反斜杠。第一个反斜杠转义第二个反斜杠，结果

---

⑤ 词“开关”和“选项”可以互换。它们是命令的参数，以一个短划线开头。

⑥ 命令 uname -n 将被执行，因为它包含在美元符后的一对圆括号中。另一种方法是将命令包含在反引号内(参见 13.12 节，“命令替换”)

是\$。这样美元符就被保护起来以防 shell 解释它，从而将它直接显示出来。

2. 给主提示符赋值为\W 和\d 转义序列，分别求出当前工作目录的基名和当天的日期。

次提示符 给 PS2 变量赋值次提示符。它的值显示在标准错误输出(默认为屏幕)上。如果没有输入完整的命令或期望更多的输入，就会出现这个提示符。默认的次提示符为>。

范例 13-14

```
1   $ echo "Hello
2   > there"
3   Hello
    there
4   $
5   $ PS2="----> "
6   $ echo 'Hi
7   ------>
    ------>
    ------> there'
    Hi


    there
    $
8   $ PS2="\s:PS2 > "
    $ echo 'Hello
    bash:PS2 > what are
    bash:PS2 > you
    bash:PS2 > trying to do?
    bash:PS2 > '
    Hello
    what are
    you
    trying to do?
    $
```

说明

1. 字符串"Hello 后面必须有一个对应的双引号。
2. 输入换行符后，出现了次提示符。如果不输入闭合双引号，次提示符就会继续出现。
3. 显示 each 命令的输出。
4. 显示主提示符。
5. 重新设置次提示符。
6. 字符串'Hi 后面必须有一个对应的单引号。
7. 输入换行符后，出现了新设置的次提示符。如果不输入闭合单引号，次提示符就会继续出现。
8. PS2 提示符被设成 shell 名(\s)后跟一个字符串，这个串包含一个冒号、PS2、>号以及一个空格。



13.2.4 搜索路径

变量 PATH 被 bash 用于定位用户在命令行键入的命令。路径是一个用冒号分隔的目录列表，shell 用这个路径来查找命令。默认路径是由系统决定的，并且由安装 bash 的管理员设定。搜索从左向右依次进行。路径末尾的点代表当前工作路径。如果在路径列出的所有目录中都未找到目标命令，shell 就会向标准错误输出发送这样一条消息：filename: not found(文件名：未找到)。如果运行的是 bash shell，那么路径通常是在.bash_profile 文件中设置。如果运行的是 sh(Bourne shell)则在.profile 文件中设置。

如果路径中未包含句点，执行当前工作目录下的命令或脚本时，必须在脚本的名字前面加上./，例如./program_name，这样脚本才能找到该程序。

范例 13-15

```
(打印 PATH)
1   $ echo $PATH
    /usr/gnu/bin:/usr/local/bin:/usr/ucb:/bin:/usr/bin:.
(设置 PATH)
2   $ PATH=$HOME:/usr/ucb:/usr:/usr/bin:/usr/local/bin:
3   $ export PATH
4   $ runit
    bash: runit: command not found
5   $ ./runit
    < program starts running here >
```

说明

1. 通过 echo$PATH 命令，显示出 PATH 变量的值。路径包括一列以冒号分隔的元素，对路径的查找是从左到右进行的。路径末尾的句点代表用户的当前工作目录。

2. 设置路径，把此列以冒号分隔的目录赋给 PATH 变量。注意，在这个路径中，句点未在路径的末尾出现可能是出于安全考虑。

3. 输出路径，让子进程也能访问它。没有必要用独占一行来输出 PATH，可以在同一行内完成多个命令，如：

export PATH=$HOME:/usr/ucb:/bin:.等

4. 因为句点不在搜索路径中，所以在当前工作目录下运行 runit 程序时，bash 找不到它。

5. 因为在程序名前加上了点和斜杠(./)，所以如果它在当前工作目录下，shell 就可以找到并执行它。

13.2.5 hash 命令

hash 命令控制系统内部的一个哈希表，shell 用这个表来提高命令查找的效率。有了这个内部哈希表，shell 就不必每输入一条命令都去搜索路径。第一次输入某条命令时，shell 通过搜索路径找到这条命令，然后将它保存在 shell 的内存空间的一个表中。再次使用同一命令时，shell 通过哈希表来查找它。这样访问命令比必须搜索整个路径要快得多。如果事先知道自己要经常使用某条命令，就可以将它加到哈希表中。也可以从这个表中删除命令。哈希命令的输出结果显示了 shell 通过该表找到某条命令的次数(hits)以及命令的完整路径

名。带-r 选项的 hash 命令将清空这个哈希表。参数--禁止选项检查其余的参数。bash 可以自动实现哈希表。我们也可以关闭它，但如果没有特殊的理由的话，建议不要这么做。

范例 13-16

```
(打印 PATH)
(命令行)
1   hash
    hits    command
    1       /usr/bin/mesg
    4       /usr/bin/man
    2       /bin/ls
2   hash -r
3   hash
    No commands in hash table
4   hash find
    hits    command
    0       /usr/bin/find
```

说明

1. hash 命令显示在这个登录会话中已经执行过的命令的完全路径名(内置命令不列出)。命中的次数就是用哈希表找到一个命令的次数。

2. 带-r 选项的 hash 命令清空哈希表中的所有记录。

3. 在上一个命令使用过-r 选项后，hash 命令报告当前表中没有命令。

4. 如果事先知道自己要经常使用某条命令，就可以将它作为 hash 命令的参数来把它加到哈希表中。find 命令已被加入到表中。现在表中显示零命中，因为命令还没有被用过。

13.2.6 source 或 dot 命令

source 命令(来自 C shell)是内置的 bash 命令。dot 命令，简单地说就是一个句点，(来自 Bourne shell)是 source 的另一个名字。两个命令都是以一个脚本名作为参数。shell 将在当前 shell 的环境中执行这个脚本，也就是说，不会为它启动一个子进程。这个脚本中设置的所有参数都将成为当前 shell 环境的一部分。source(或 dot)命令通常被用来重新执行经过修改的初始化文件.bash_profile、.profile 等。例如，如果在登录之后修改了.bash_profile 中某项设置，比如变量 EDITOR 或 TERM，用 source 命令重新执行.bash_profile 就可以让修改生效。而不必先注销再重新登录进来。像.bash_profile 文件或其他类似的 shell 脚本不需要执行权限就可以用 source 和 dot 命令来执行。

范例 13-17

```
$ source .bash_profile
$ . .bash_profile
```

说明

source 和 dot 命令在当前 shell 中执行初始化文件.bash_profile。局部和全局变量都将在



当前 shell 中重新定义。dot 命令免去了先注销再重新登录回来的麻烦[7]。

13.3 命令行

用户登录成功后，shell 会显示它的主提示符，默认情况下是一个美元符。shell 就是命令解释器。以交互方式运行时，shell 从终端读取命令，把命令行拆分为若干个词。命令行由一或多个词(标记)组成，词之间以空白符(空格或制表符)分隔、以换行符结束，换行符则是通过按下回车键产生的。命令行的第一个词是命令，后跟命令的参数。命令可以是一个 UNIX/Linux 可执行程序(比如 ls 和 date)，也可以是 shell 的一条内置命令(比如 cd 和 pwd)或某个 shell 脚本。命令可能含有称作元字符的特殊字符，shell 分析命令行时必须解释这些元字符。如果命令行很长，需要转到下一行继续输入，就必须先输入一个反斜杠，然后再换行，这样才能在下一行接着输入。命令行结束之前，shell 会在接下来的每一行上显示次提示符。

13.3.1 处理命令的顺序

命令行的第一个词就是将要执行的命令。命令可能是关键词、别名、函数、特定的内置命令或应用程序、可执行程或 shell 脚本。命令将根据其类型按以下顺序执行。

(1) 别名

(2) 关键字(如 if、function、while、until)

(3) 函数

(4) 内置命令

(5) 可执行文件和脚本

特定的内置命令和函数是在 shell 中定义的，因此在当前 shell 的环境中运行它们将快得多。像 ls 和 date 这样的脚本和可执行程序是存储在磁盘上的，shell 为了运行它们，必须首先通过搜索 PATH 环境变量中的目录来查找它们。然后 shell 调用一个新的 shell 来执行脚本。可以使用内置的 type 命令来查看命令的类型——内置命令、别名、函数、可执行文件等(请参见范例 13-18)。

范例 13-18

```
$ type pwd
pwd is a shell builtin
$ type test
test is a shell builtin
$ type clear
clear is /usr/bin/clear
$ type m
m is aliased to 'more'
```

[7] 如果把.bash_profile 作为脚本直接运行，就会启动一个子 shell。结果是在子 shell，而不是登录 shell(即父 shell)中设置变量。当子 shell 退出时，父 shell 将不再有它的任何设置。

```
$ type bc
bc is /usr/bin/bc
$ type if
if is a shell keyword
$ type -path cal
/usr/bin/cal
$ type which
which is aliased to 'type -path'
$ type greetings
greetings is a function
greetings ()
{
      echo "Welcome to my world!";
}
```

13.3.2 内置命令和 help 命令

内置命令是 shell 内部源代码的一部分。它们是内置的，且很容易被 shell 获取，而像 date、cal 和 finger 这样的命令是经过编译的二进制程序且驻留在磁盘上。执行一个内置命令消耗很少，因为它不涉及磁盘操作。和磁盘上的程序相比，shell 先执行内置命令。bash 已经新增加了一个在线帮助系统，这样可以查看所有的内置命令，或一个特定内置命令的描述，help 本身就是一个内置命令。请参见 13.20 节中的表 13-28，它列出了所有的内置命令。

范例 13-19

```
1  $ help help
   help: help [pattern ...]
   Display helpful information about built-in commands. if PATTERN
   is specified, gives detailed help on all commands matching
   PATTERN, otherwise a list of the built-ins is printed.
2  $ help pw
   pwd: pwd
      Print the current working directory.
```

13.3.3 改变命令行处理的顺序

bash 提供了 3 个内置命令可以忽视命令行处理的顺序：command、builtin 和 enable。

command 内置命令将别名和函数从处理顺序中去掉。只处理搜索路径中的内置命令和可执行程序。

builtin 命令只查找内置命令，忽略在路径中找到的函数和可执行程序。

内置命令 enable 可以打开和关闭内置命令。默认时，内置命令是打开的。关闭了一个内置命令后，磁盘上和内置命令同名的可执行命令无需指定全路径名就可以执行(在正常的处理情况下，bash 首先搜索内置命令，然后再搜索可执行命令)。使用-n 开关可以关闭内置命令。例如，对于新 shell 程序员来说，最容易混淆的就是给一个脚本命名为 test。因为 test 是一个内置命令，shell 将去执行这个内置命令而不是用户的脚本(内置命令通常在可执行程序前执行)。通过键入：enable –n test，关闭 test 内置命令，用户的脚本将优先执行。



不带任何选项，enable 内置命令将列出所有的内置命令。下面列出的每个内置命令将在 13.20 节，“shell 内置命令”中进行介绍。

范例 13-20

```
1   $ enable
    enable .
    enable :
    enable [
    enable alias
    enable bg
    enable bind
    enable break
    enable builtin
    enable cd
    enable command
    enable continue
    enable declare
    enable dirs
       .....
    enable read
    enable readonly
    enable return
    enable set
    enable shift
    enable shopt
       .....
    enable type
    enable typeset
    enable ulimit
    enable umask
    enable unalias
    enable unset
    enable wait
2   enable -n test
3   function cd { builtin cd; echo $PWD; }
```

说明

1. 不带任何选项的 enable 内置命令将显示完整的 bash shell 内置命令列表。该范例只列出了其中的一部分。

2. 使用-n 开关，test 内置命令被关闭。现在，可以执行名为 test 的脚本，而无需担心执行的是内置命令 test。不推荐以操作系统命令的名字来命令一个脚本，因为试图在另外一个 shell 中运行这个同名脚本时，内置命令并没有被关闭。

3. 函数名是 cd。builtin 使得内置命令 cd 包含在函数的定义中，且调用时代替函数 cd，以防出现无限递归循环。

13.3.4 退出状态

命令或程序终止后，会向父进程返回一个退出状态。退出状态是一个 0~255 之间的整数。按照惯例，程序退出时，如果返回的状态是 0，表示命令执行成功。如果退出状态非 0，则表示命令因某种原因而执行失败。如果 shell 没有找到命令，返回的状态是 127。如果是

一个致命的信号导致命令终止，则退出状态是 128 并加上导致它终止的信号编号。

shell 的状态变量?被设置为 shell 执行的上一条命令的退出状态值。这样，程序运行结果是成功还是失败，将由编写它的程序员来判断。

范例 13-21

```
1   $ grep ellie /etc/passwd
    ellie:MrHJEFd2YpkJY:501:501::/home/ellie:/bin/bash
2   $ echo $?
    0
3   $ grep nicky /etc/passwd
4   $ echo $?
    1
5   $ grep ellie /junk
    grep: /junk: No such file or directory
6   $ echo $?
    2
7   $ grip ellie /etc/passwd
    bash: grip: command not found
8   $ echo $?
    127
9   $ find / -name core ^C    # User presses Ctrl-C
10  $ echo $?
    130
```

说明

1. grep 命令在文件/etc/passwd 中查找模式 ellie，并且成功。grep 显示从/etc/passwd 中找到的行。
2. 变量?被设置为 grep 命令的退出状态值。值为 0 表示成功。
3. grep 命令在文件/etc/passwd 中找不到用户 nicky。
4. grep 程序如果找不到模式，变量?的返回值为非 0。退出状态 1 表示失败。
5. grep 因为打不开文件/junk 而运行失败。grep 将错误信息发送到标准输出，即屏幕上。
6. grep 如果找不到文件，就会返回退出状态 2。
7. shell 没有找到 grip 命令。
8. 因为命令没有找到，返回退出状态 127。
9. 按下 Ctrl+C 组合键发出的 SIGINT 信号终止了 find 命令。Ctrl+C 的信号编号是 2。
10. 从被终止的进程返回的状态是 128+信号编号。即 128+2。

13.3.5 含多条命令的命令行

一个命令行可以包含多条命令。命令之间用分号隔开，命令行以换行符终止。退出状态是一列命令中的最后一条命令。

范例 13-22

```
$ ls; pwd; date
```

说明

从左到右逐一执行命令，直至遇到换行符。



13.3.6 命令编组

可以把多条命令合为一组，这样就能将所有命令的输出通过管道发给另一条命令，或者重定向到某个文件。

范例 13-23

```
$ ( ls; pwd; date ) > outputfile
```

说明

每条命令的输出都被发送到文件 outputfile。圆括号内侧的空格是必需的。

13.3.7 命令的条件执行

有条件地执行命令时，要用特殊的元字符，即双与号(&&)或双竖杠(||)分隔两个命令串。是否执行这两个元字符右侧的命令取决于左侧命令的退出状态。

范例 13-24

```
$ cc prgm1.c -o prgm1 && prgm1
```

说明

如果第一条命令执行成功(退出状态为 0)，就执行&&后面的命令。即：如果 cc 程序能成功编译 prgm1.c，就执行它生成的可执行程序 prgm1。

范例 13-25

```
$ cc prog.c >& err || mail bob < err
```

说明

如果第一条命令执行失败(退出状态不为 0)，就执行| |后面的命令。即：如果 cc 程序未能成功编译 prgm1.c，就把报错信息写到文件 err 中，然后通过邮件将 err 发给用户 bob。

13.3.8 在后台执行的命令

执行命令时，命令通常都在前台运行，要等到命令执行完之后，提示符才会重新出现。等待命令结束有时会不太方便。如果在命令行末尾加上一个与号(&)，shell 就会立即返回 shell 提示符，同时在后台执行这条命令。于是，不需要等待就能启动另一个程序。后台任务会在执行过程中将它产生的输出随时发送到屏幕上。因此，如果想在后台运行一条命令，不妨将它输出重定向到某个文件，或者通过管道发给某个设备(比如打印机)，这样，后台命令的输出结果就不会干扰正在前台进行的工作。

变量!保存最后一个进入后台的作业的 PID 号(有关后台处理的详细内容请参见 13.4 节的“作业控制”)。

范例 13-26

```
1   $ man sh | lp&
2   [1] 1557
3   $ kill -9 $!
```

说明

1. 通过管道将 man 命令的输出(即 UNIX 命令的手册页)发给打印机。命令行末尾的与号把作业置入后台。

2. 屏幕上显示了两个数字：方括号里的数字说明这是第一个被放入后台的作业。第二个数是一个 PID，是该作业的进程标识号。

3. shell 提示符立刻就出现了。程序在后台运行时，shell 正等待着下一条在前台运行的命令。变量!的值是最近那个被放入后台的作业的 PID。如果能及时获取到这个值，就能赶在作业进入打印队列之前终止它。

13.4 作业控制

作业控制是 bash shell 的一项强大功能，可以选择在后台或前台运行作业。一个正在运行的程序称为进程或作业，每个进程有一个进程标识号，即 PID。通常，在命令行输入的命令都在前台运行，并且持续运行于前台直至结束，除非按下 Ctrl+C 或 Ctrl+\组合键来发送信号终止它。通过作业控制，可以将一个作业置于后台运行，可以通过 Ctrl+D 组合键暂停一个作业，这样作业将被发送到后台并挂起，可以使一个暂停的作业在后台运行，可以将一个后台作业送回前台，甚至还可以终止已经在后台或前台运行的作业。本节最后的表 13-3 列出了作业命令的清单。

作业控制命令和选项

默认情况下，作业控制总是设置的(一些老版本的 UNIX 不支持这种功能)，如果系统没有设置，可以通过下面的命令来重新设置：

格式

```
set -m            # set job control in the .bashrc file
set -o monitor    # set job control in the .bashrc file
bash -m -i        # set job control when invoking interactive bash
```

范例 13-27

```
1   $ vi
    [1]+  Stopped     vi
2   $ sleep 25&
    [2] 4538
3   $ jobs
    [2]+  Running     sleep 25&
    [1]-  Stopped      vi
4   $ jobs -l
    [2]+ 4538  Running      sleep 25&
    [1]- 4537  Stopped       vi
5   $ jobs %%
    [2]+ 4538  Running      sleep 25&
6   $ fg %1
7   $ jobs -x echo %1
```

```
    4537
8   $ kill %1        # or kill 4537
    [1]+  Stopped      vi
    Vim: Caught deadly signal TERM
    Vim: Finished.
    [1]+  Exit 1      vi
```

说明

1. 调用 vi 编辑器后，可以按下^Z(Ctrl+Z 组合键)将 vi 会话暂停。编辑器将在后台被挂起，消息“Stopped”后，将立即出现 shell 提示符。

2. 命令末尾的&号使参数为 25 的 sleep 命令在后台执行。标记[2]表明这是第 2 个运行在后台的作业，它的 PID 是 4538。

3. jobs 命令显示当前在后台的作业。

4. 带-l 选项的 jobs 命令显示运行在后台的进程(作业)和它们的 PID 号。

5. %%参数使 jobs 显示最近一条放入作业表中的命令。

6. fg 命令后跟一个百分号和作业号将把该作业调到前台。没有作业号，fg 将把最近放到后台的作业调回前台。

7. －x 选项只打印作业的 PID 号。%1 代表在第一个例子中暂停的 vi 会话。

8. kill 命令发送一个 TERM 信号给进程并终止它。vi 程序被终止。可以指定作业号或 PID 号作为 kill 命令的参数。

表 13-3 作业控制命令

命　令	含　义
bg	启动被终止的后台作业
fg	将后台作业调到前台
jobs	列出所有正在运行的作业
kill	向指定作业发送 kill 信号
stop	挂起一个后台作业
stty tostop	当一个后台作业向终端发送输出时就挂起它
wait [n]	等待一个指定的作业并返回它的退出状态，这里 n 是一个 PID 或作业号
^Z(Ctrl-Z)	终止(挂起)作业。屏幕上将出现提示符

jobs 命令的参数	含　义
%n	作业号 n
%string	以 string 开头的作业名
%?string	作业名包含 string
%%	当前作业
%+	当前作业
%-	当前作业前的一个作业
-r	列出所有运行的作业
-s	列出所有挂起的作业

新的作业选项 在 bash 2.x 版本中给 jobs 命令增加了两个新选项。它们是-r 和-s 选项。-r 选项列出所有运行的作业，-s 选项列出所有暂停的作业。

disown 内置命令 disown 内置命令(bash 2.x)从作业表中删除一个指定的作业。在作业被删除后，shell 将不再认为它是一个可行的作业进程，且只通过它的进程 ID 号来访问它。

13.5 命令行快捷方式

13.5.1 命令和文件名补全

为了减少键入，bash 实现了命令和文件名补全机制，它允许键入命令名或文件名的一部分，然后按下 Tab 键，这时它将补上该命令名或文件名的剩余部分。

如果键入命令的前几个字母并按下 Tab 键，bash 将尝试补全这个命令并执行它。如果 bash 因为文件或命令不存在而不能扩展它们时，终端将发出蜂鸣声且光标将停在命令的末尾。如果有不止一条命令是以这几个字符开头的，并且再一次按下了 Tab 键，那么将列出所有以这几个字符开头的命令。

如果有几个文件都是以同样的字母开头的，bash 将选择所匹配的最短名字来扩展文件名直至字符不同，然后闪烁光标提示您完成剩余部分。

范例 13-28

```
1   $ ls
    file1 file2 foo foobarckle fumble
2   $ ls fu[tab]         # Expands filename to fumble
3   $ ls fx[tab]         # Terminal beeps, nothing happens
4   $ ls fi[tab]         # Expands to file_  (_ is a cursor)
5   $ ls fi[tab][tab]    # Lists all possibilities
    file1 file2
6   $ ls foob[tab]       # Expands to foobarckle
7   $ da[tab]            # Completes the date command
    date
    Tue Feb 24 18:53:40 PST 2004
8   $ ca[tab][tab]       # Lists all commands starting with ca
    cal   captoinfo  case   cat
```

说明

1. 列出当前工作目录下的所有文件。

2. 键入 fu 后，按下 Tab 键，补全文件名拼写出 fumble 并显示。

3. 因为没有文件以 fx 开头，终端将发出蜂鸣声，且光标停留在原处，但什么也不做(如果这个特性已经被关闭则终端不会发出蜂鸣声)。

4. 列出以 fi 开头的许多文件；扩展文件名直至不再有相同的字母。如果再次按下 Tab 键，将列出所有包含该拼写的文件。

5. 通过两次按下 Tab 键，打印出所有以 file 开头的文件。

6. 按下 Tab 键，文件名补全为 foobarckle。



7. 当在 da 后按下 Tab 键时，以 da 开头的命令只有 date。命令名被补全并执行。

8. 当在 ca 后按下 Tab 键时，什么也没发生，因为有不止一个命令以 ca 开头。再次按下 Tab 键将列出所有以 ca 开头的命令。

13.5.2 历史

bash shell 提供命令行历史机制。它将在命令行键入的命令保存为一个带编号的清单。在一个登录会话中，键入的命令会保存在 shell 内存中的历史清单中，并在退出时添加到历史文件中。您可以从历史列表中调出某条命令再次执行，而不必重新输入这条命令。内置命令 history 可用来显示历史清单。默认的历史文件名是.bash_history，它位于主目录中。

当 bash 开始访问历史文件时，HISTSIZE 变量指明有多少条命令可以从历史文件复制到历史清单中。默认大小是 500。HISTFILE 变量指明保存命令的命令行历史文件的名称(默认是~/.bash_history)。如果没有设置，当一个交互式 shell 退出时就不会保存命令行历史。

从一个登录会话到另一个的过程中，历史文件的行数会相应地增长。HISTFILESIZE 变量控制历史文件能包含的最大行数。如果给这个变量赋了值，当历史文件的行数超出该值时将移出最靠前的行。最大行数的默认值为 500。

fc –l 命令可以用来显示或编辑历史清单中的命令。

表 13-4 历史变量

FCEDIT	使用 fc 命令的 UNIX/Linux 编辑器的路径名
HISTCMD	当前命令的历史编号，或在历史清单中的序号。若未设置，它的特性将丢失。即使之后进行重置也将不起用用
HISTCONTROL	如果设置了 ignorespace 的值，以空格开头的行将不会进入历史清单。如果设置了 ignoredups 的值，和最后一个历史行匹配的行不会进入。ignoreboth 的值结合了两个选项。如果没有设置，或设成了上面两个值以外的其他值，那么解释器读到的所有行都将保存在历史清单中
HISTFILE	指定保存命令行历史的文件。默认是~/.bash_history。如果没有设置，当一个交互式 shell 退出时不会保存命令行历史
HISTFILESIZE	历史文件能包含的最大行数。当给这个变量赋值后，如果有必要，历史文件将被截尾，以使包含的行数不超过这个数。默认值是 500
HISTIGNORE	以冒号分隔的一系列模式，用来决定哪些命令行应该保存在历史清单中。每个模式确定行的开始且包括正常的 shell 匹配字符模式。用于模式中的&使 history 命令忽略重复。例如，ty??:&将匹配任何以 ty 开头后跟两个字符的命令行，以及该命令的重复。这些命令不会放入历史清单中
HISTSIZE	记录在命令历史中的命令数。默认值是 500

13.5.3 从历史文件访问命令

方向键 从历史文件中访问命令，可以用方向键在历史文件中上下左右移动(见表

13-5)。可以在历史文件中编辑任意行，用标准键来删除、修改、退格等。当编辑完一行后，只要按下 Enter 键就会执行该命令行。

表 13-5 方向键

↑	向上箭头在历史清单中向上移动
↓	向下箭头在历史清单中向下移动
→	右箭头使光标在历史命令中向右移动
←	左箭头使光标在历史命令中向左移动

history 内置命令 history 内置命令显示键入的历史命令，每条命令前对应一个事件号。

范例 13-29

```
1  $ history
   982  ls
   983  for i in 1 2 3
   984  do
   985  echo $i
   986  done
   987  echo $i
   988  man xterm
   989  adfasdfasdfadfasdfasdfadfasdfasdf
   990  id -gn
   991  id -un
   992  id -u
   993  man id
   994  more /etc/passwd
   995  man ulimit
   996  man bash
   997  man baswh
   998  man bash
   999  history
   1000  history
```

说明

内置的 history 命令显示历史清单中记录的命令。

fc 命令 fc 命令，也称为 fix 命令，可以用于两个方面：(1)从历史清单中选择命令(2)用 vi 或 emacs 编辑器，或系统上的任何其他编辑器来编辑命令。

第一种格式，带-l 选项的 fc 可以从历史清单中选择指定的行或行的范围。当-l 选项打开时，输出发送到屏幕上。例如，fc –l，默认地打印历史清单中最后 16 行，fc –l 10 选择从编号 10 到清单末尾的所有行，fc –l –3 选择最后三行。-n 开关关闭历史清单中命令的编号。这个选项打开时，可以选择一段范围内的命令并将它们重定向到一个文件中，该文件会被当作一个 shell 脚本依次执行。-r 开关反转命令的序号。

fc 的第二种格式将在 13.5.4 节，“命令行编辑”中介绍。



表 13-6 fc 命令

fc 参数	含 义
-e editor	将历史清单调入编辑器
-l n-m	列出编号从 n 到 m 的命令
-n	关闭历史清单的编号
-r	反转历史清单的序号
-s string	访问以 string 开头的命令

范例 13-30

```
1  $ fc -l
    4       ls
    5       history
    6       exit
    7       history
    8       ls
    9       pwd
   10       clear
   11       cal 2000
   12       history
   13       vi file
   14       history
   15       ls -l
   16       date
   17       more file
   18       echo a b c d
   19       cd
   20       history
2  $ fc -l -3
   19       cd
   20       history
   21       fc -l
3  $  fc -ln
        exit
        history
        ls
        pwd
        clear
        cal 2000
        history
        vi file
        history
        ls -l
        date
        more file
        echo a b c d
        cd
        history
```

```
        fc -l
        fc -l -3
4 $ fc -ln -3 > saved
5 $ more saved
        fc -l
        fc -l -3
        fc -ln
6 $ fc -l 15
  15      ls -l
  16      date
  17      more file
  18      echo a b c d
  19      cd
  20      history
  21      fc -l
  22      fc -l -3
  23      fc -ln
  24      fc -ln -3 > saved
  25      more saved
  26      history
7 $ fc -l 15 20
  15      ls -l
  16      date
  17      more file
  18      echo a b c d
  19      cd
  20      history
```

说明

1. fc -l 列出历史清单中最后 16 条命令。
2. fc -l -3 从历史清单中选择最后 3 条命令。
3. 带-ln 选项的 fc 打印出没有行号的历史清单。
4. 历史清单中的最后 3 条命令，不带行号，被重定向到文件 saved 中。
5. 显示文件 saved 的内容。
6. 列出历史清单中从 15 行开始的命令。
7. 显示行号为 15~20 的命令。

给 fc 带上-s 选项，可以用一个串模式来重新执行过去的一条命令。比如，fc –s rm 将使包含模式 rm 的最近行被重新执行。可以创建一个 shell 别名 r 来模仿 Korn shell 的 redo 命令，如 alias r='fc –s'，如果在命令行键入 r vi，包含模式的最近一条历史记录将被重新执行。这样，vi 编辑器就如最近一次被启动时一样，包含了传进的任何参数。

范例 13-31

```
1   $ history
    1   ls
    2   pwd
    3   clear
    4   cal 2000
```




```
    5   history
    6   ls -l
    7   date
    8   more file
    9   echo a b c d
2   $ fc -s da
    date
    Thu Jul  15 12:33:25 PST 2004
3   $ alias r="fc -s"
4   $ date +%T
    18:12:32
5   $ r d
    date +%T
    18:13:19
```

说明

1. 内置的 history 命令显示历史清单。

2. 带-s 选项的 fc 命令搜索最近以 da 开头的命令。在历史清单中找到 date 命令并重新执行它。

3. 给别名(用户定义的简称)r 赋值命令 fc －s。这意味着任何时候在命令行键入 r，它都将被 fc －s 替换。

4. 执行 date 命令。它将打印出当前时间。

5. 用别名来作为 fs －s 命令的快捷方式。最近一条以 d 开头的命令将被重新执行。

重新执行历史命令(bang!bang!)　重新执行历史清单中的命令需要使用感叹号(称为 bang)。如果键入两个感叹号(!!)，即“bang”、“bang”历史清单中的最后一条命令将重新执行。如果键入一个感叹号，后跟一个数字，那么以该数字列出的命令将重新执行。如果键入一个感叹号和一个字母或字符串，那么最近一条以该字母或字符串开头的命令将重新执行。脱字符(^)也可以用来作为编辑过去命令的快捷方式。表 13-7 列出了所有的历史替换字符。

表 13-7　替换和历史

事件指示者	含　义
!	说明开始历史替换
!!	重新执行上一条命令
!N	重新执行历史清单中的第 N 条命令
!-N	重新执行从当前命令往回数的第 N 条命令
!string	重新执行最后一条以串 string 开头的命令
!?string?	重新执行最后一条包含串 string 的命令
!?string?%	重新执行历史清单中最近一条包含串 string 的命令行参数
!$	用上一条命令的最后一个参数作为当前命令行
!! string	将 string 添加到上一条命令的最后并执行
!N string	将 string 添加到历史清单中第 N 条命令的最后并执行

(续表)

事件指示者	含　义
!N:s/old/new/	在前面的第 N 条命令中，将第一次出现的 old 串替换成 new 串
!N:gs/old/new/	在前面的第 N 条命令中，将所有的 old 串替换成 new 串
^old^new^	在上一条命令中，用 new 串替换 old 串
Command !N:wn	在当前命令后添加一个来自前第 N 条命令的参数(wn)并执行它。wn 是一个从 0，1，2，...开始的数，表明前面那个命令的第几个词。词 0 是命令本身，1 是它的第一个参数等(请参见范例 13-32)

范例 13-32

```
1   $ date
    Mon Jul  12 12:27:35 PST 2004
2   $ !!
    date
    Mon Jul  12 12:28:25 PST 2004
3   $ !106
    date
    Mon Jul  12 12:29:26 PST 2004
4   $ !d
    date
    Mon Jul  12 12:30:09 PST 2004
5   $ dare
    dare: Command not found.
6   $ ^r^t
    date
    Mon Jul  12 12:33:25 PST 2004
```

说明

1. 在命令行执行中 UNIX/Linux 的 date 命令。历史清单随之被更新，date 成了清单中的最后一条命令。

2. !!(bang bang)从历史清单中取出最后那条命令，这条命令再次被执行。

3. 执行历史清单中第 106 条命令。

4. 执行命令清单中最后那条以字母 d 开头的命令。

5. 敲错了命令，应该是 date，而不是 dare。

6. 用脱字符在历史清单的最后那条命令中替换字母。命令中第一个 r 被替换为 t。即 dare 改为 date。

范例 13-33

```
1   $ ls  file1 file2 file3
    file1 file2 file3
    $ vi !:1
    vi file1
2   $ ls file1 file2 file
    file1 file2 file3
    $ ls !:2
```



```
    ls file2
    file2
3   $ ls file1 file2 file3
    $ ls  !:3
    ls file3
    file3
4   $ echo a b c
    a b c
    $ echo !$
    echo c
    c
5   $ echo a b c
    a b c
    $ echo !^
    echo a
    a
6   % echo a b c
    a b c
    % echo !*
    echo a b c
    a b c
7   % !!:p
    echo a b c
```

说明

1. ls 命令列出 file1、file2 和 file3。历史列表被更新，命令行被分解为若干个词，词的编号从 0 开始。如果在词的编号前加个冒号，就能将这个词从历史清单中提取出来。标记!:1 的含义是：取出历史清单中最后那条命令的第一个参数，并用它替换掉命令串中的!:1。最后那条命令的第一个参数是 file1(第 0 号单词是命令本身)。

2. !:2 被替换为上一条命令的第 2 个参数，即 file2，并且成为 ls 的参数。命令运行结果是打印出 file2(file2 是第 3 个单词)。

3. ls !:3 的含义是：找到历史清单中的最后一条命令，取出该命令中第 4 个词，把它作为参数传给 ls 命令(file3 是第 4 个词)。

4. 带美元符($)的感叹号(!)代表历史清单中最后那条命令的最后一个参数。此时这个参数是 c。

5. 脱字符(^)代表命令后的第一个参数。后跟脱字符的感叹号(!)代表历史清单中最后那条命令的第一个参数。本例中最后那条命令的第一个参数是 a。

6. 星号(*)代表命令后的所有参数。感叹号(!)后跟一个星号，代表历史清单中最后那条命令的所有参数。

7. 显示历史清单中最后一条命令，但不执行。历史清单被更新。现在可以对该行执行脱字符替换。

13.5.4 命令行编辑

bash shell 提供两个内置的编辑器，emacs 和 vi，可以用来交互地编辑历史清单。在命令行使用编辑特性时，不管是在 vi 还是 emacs 模式下，都由 readline 函数决定按键所完成的功能。例如，若在 emacs 中，使用 Ctrl+P 组合键可以在命令行历史中上移，若在 vi 中，

K 键在历史清单中上移。readline 还可以控制方向键、光标移动、更改、删除和插入文本，以及恢复键入和撤消键入。readline 的另一个特性是在 13.5.1“命令和文件名补全”里讨论过的补全特性。可以键入命令或文件名的一部分，然后按下 Tab 键，剩余的部分由它的补全功能补全。readline 库还提供了许多特性以帮助在命令行操作文本。

emacs 内置编辑器是默认的内置编辑器且是非模态的，而 vi 内置编辑器工作在两种模式下，一种是执行命令行上的命令，另一种是进入文本。若使用过 UNIX，那大概至少对其中的一种编辑器比较熟悉。为了使 vi 编辑器可用，增加下面列出的 set 命令[⑧]并将该行放到~/.bashrc 文件中。为了设置 vi，在提示符或~/.bashrc 文件中键入下面例子中的内容。

范例 13-34

```
set -o vi
```

说明

为历史清单的命令行编辑设置 vi 内置编辑器。

若要换成 emacs 编辑器，则键入：

范例 13-35

```
set -o emacs
```

说明

为历史清单的命令行编辑设置 emacs 内置编辑器。

vi 内置编辑器　要编辑历史清单，在命令行按下 Esc 键。如果想在历史清单中向上移动的话，按下 K 键，按下 J 键[⑨]可以向下移动，就像标准的 vi 移动键。当找到想编辑的命令时，可以使用 vi 的标准键左右移动，删除、插入或更改文本(请参见表 13-8)。编辑完成后，按下 Enter 键。该命令将被追执行并被加到历史清单的末尾。

表 13-8　vi 命令

命　　令	功　　能
在历史文件中移动：	
ESC k 或+	上移历史清单
ESC j 或-	下移历史清单
G	移到历史文件的第一行
5G	移到历史文件中第 5 行
/string	向上搜索历史文件查找串 string
?	向下搜索历史文件查找串
在一行中移动：	
h	在一行中左移
l	在一行中右移

⑧　如果没有设置 set -o(编辑器)，但 EDITOR 变量被设成 emacs 或 vi，则 bash 将会使用这个定义。

⑨　vi 是区分大小写的。大写 J 和小写 j 对应不同的命令。



(续表)

命　　令	功　　能
b	向后移动一个词
e 或 w	向前移动一个词
^或 0	移到行首
$	移到行尾
用 vi 编辑：	
a A	添加文本
i I	插入文本
dd dw x	删除文本到缓存中(行、词或字符)
cc C	修改文本
u U	撤消
yy Y	移出(把一行复制到缓存)
p P	把移出或删除的行放在该行下面或上面
r R	在一行中替换一个字母或任意大小的文本

内置 emacs 编辑器　要想使用 emacs 内置编辑器，像 vi 一样，在命令行启动它。要沿历史文件向上移动，就按 Ctrl+P 组合键。要向下移动，就按 Ctrl+N 组合键。用 emacs 编辑命令来修改或纠正文本，然后按下 Enter 键，将重新执行该命令。参见表 13-9。

表 13-9　emacs 命令

命　　令	功　　能
Ctrl+P	上移历史文件
Ctrl+N	下移历史文件
Ctrl+B	向后移一个字符
Ctrl+R	向后搜索串
Esc B	向后移一个词
Ctrl+F	向前移一个字符
Esc F	向前移一个词
Ctrl+A	移到行首
Ctrl+E	移到行尾
Esc <	移到历史文件的第一行
Esc >	移到历史文件的最后一行
用 emacs 编辑：	
Ctrl+U	删除行
Ctrl+Y	恢复行
Ctrl+K	从光标处一直删除到行尾
Ctrl+D	删除一个字母
Esc D	向前删除一个词

(续表)

命　令	功　能
Esc H	向后删除一个词
Esc space	在光标处设一个标志
Ctrl+X Ctrl+X	交换光标和标志
Ctrl+P Ctrl+Y	把从光标到标志之间的区域放到一个缓存中(Ctrl+P)并记下来(Ctrl+Y)

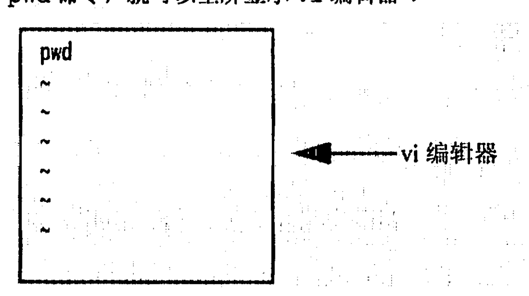
PCEDIT 和编辑命令　如果带-e 选项的 fc 命令后跟一个 UNIX 编辑器的名称，编辑器会被调用，包括从历史清单中选择的命令。如，fc –e vi –l –3 将调用 vi 编辑器，在/tmp 中创建一个临时文件，把自历史清单中的最后三条命令放在编辑器的缓存中。命令可以被编辑或注释掉(在命令前加一个#号可以将其注释掉)。如果用户退出编辑器，命令将回显且被执行[⑩]。

如果没有给定编辑器名，将使用 FCEDIT 变量的值(通常在初始化文件 bash_profile 或.profile 中设置)，若没有设置 FCEDIT，则使用 EDITOR 变量的值。编辑完成并退出编辑器时，所有编辑过的命令都将回显并执行。

范例 13-36

```
1   $ FCEDIT=/bin/vi
2   $ pwd
3   $ fc
```

<使用第一行提供的 pwd 命令，就可以全屏显示 vi 编辑器 >




```
4   $ history
    1 date
    2 ls -l
    3 echo "hello"
    4 pwd
5   $ fc -3 -1        # Start vi, edit, write/quit, and execute
                      # last 3 commands.
```

说明

1. FCEDIT 变量可以赋值为系统中任何 UNIX/Linux 文本编辑器(如 vi、emacs 等)的路径名。如果没有设置，vi 编辑器为默认值。

2. 在命令行键入 pwd 命令。它将会被放在 history 文件中。

⑩ 无论用户是保存退出，还是直接退出，命令都将被执行，除非它们被注释掉或删除。



3. fc 命令调用编辑器(在 FCEDIT 中设置)，并显示所键入的上一条命令。如果用户完成键入并退出编辑器，则将执行在那里键入的任何命令。

4. history 命令列出最近键入的命令。

5. fc 命令用来启动编辑器，并把 history 文件中的最后三条命令放在编辑器的缓存中。

13.6 别名

别名是 bash shell 中用户自定义的命令简写形式。当某条命令要带多个选项和参数，或者命令语法很难记住时，别名就变得很有用。在命令行设置的别名不会被子 shell 继承。别名的设置通常在文件.bashrc 中进行。每个新 shell 启动时都要执行.bashrc 文件，所以，该文件中设置的所有别名都会为新 shell 复位。别名可以传递给 shell 脚本，但是这样会导致潜在的移植问题，除非所用的别名是直接定义在脚本中的。

13.6.1 列出别名

内置命令 alias 能够列出所有已设置的别名。输出时先打印别名，然后是它代表的实际命令或命令组。

范例 13-37

```
$ alias
alias co='compress'
alias cp='cp -i'
alias mroe='more'
alias mv='mv -i'
alias ls='ls --colorztty'
alias uc='uncompress'
```

说明

alias 命令把命令的别名(简称)列在第一列，第二列中则是别名及其代表的实际命令。

13.6.2 创建别名

alias 命令也可用来创建别名。第一个参数是别名的名字，即命令的绰号。该行的剩余部分包括一条或多条命令，这些命令将在执行别名时被执行。bash 的别名不处理参数(请参见 13.16.1 节“定义函数”)。多条命令之间用分号分隔，包含空格或元字符的命令必须用单引号括起来。

范例 13-38

```
1   $ alias m=more
2   $ alias mroe=more
3   $ alias lF='ls -alF'
4   $ alias r='fc -s'
```

说明

1. 给 more 命令设了一个别名：m。

2. 把 more 命令的别名设为 mroe。这样能方便那些容易拼写出错的人。

3. 由于命令中包含空白字符，所以被括在单引号中。别名 lF 是命令 ls -alF 的简称。

4. 别名 r 将用来代替 fc -s，它根据一个指定的模式从历史清单中查找命令。如，r vi 将重新执行历史清单中最后一条包含模式 vi 的命令。

13.6.3 删除别名

unalias 命令用来删除别名。若要暂时关闭一个别名，可以在别名的名字前加上一个反斜杠。

范例 13-39

```
1   $ unalias mroe
2   $ \ls
```

说明

1. unalias 命令从定义别名的列表中删除别名 mroe。

2. 为了执行真正的 ls 命令而将别名 ls 的含义暂时关闭。

13.7 操作目录栈

工作时，经常在目录树中用 cd 命令切换许多相同的目录，我们可以将这些目录压入一个目录栈并对目录栈进行操作而简化对它们的访问。pushd 内置命令将目录压入栈而 popd 命令则将它们弹出(请参见例 13-40)。目录栈是一系列的目录，最左边的是最近被压入栈的目录。可以用内置命令 dirs 列出所有的目录。

13.7.1 内置命令 dirs

带-l 选项的内置命令 dirs，将以完全路径名的格式显示目录栈中的所有目录。不带任何选项的 dir 用一个代字符号来表示主目录。带一个+n 选项，dirs 显示目录列中从左数起第 n(从 0 开始)个目录项。带一个-n 选项，dirs 完成同样的事情，但是从右边开始数起第 n 个。

13.7.2 pushd 命令和 popd 命令

带一个目录作为参数的 pushd 命令将这个新目录加到目录栈中，且同时切换到那个目录。如果参数是+n(这里 n 是一个数)，pushd 将旋转栈，这样从最左边开始栈中的第 n 的目录将被放到栈顶。如果参数是-n，它完成同样的工作，只是从最右边开始。不带任何参数，pushd 将交换栈顶的两项，这样使得来回切换两个目录很方便。

popd 命令从栈项删除一个目录并切换到那个目录。带+n 选项(这里 n 是一个数)，popd 将删除 dirs 命令所示的列中从最左边开始的第 n 项。



范例 13-40

```
1  $ pwd
   /home/ellie
   $ pushd ..
   /home ~
   $ pwd
   /home
2  $ pushd      # Swap the two top directories on the stack
   ~ /home
   $ pwd
   /home/ellie
3  $ pushd perlclass
   ~/perlclass  ~  /home
4  $ dirs
   ~/perlclass ~ /home
5  $ dirs -l
   /home/ellie/perlclass /home/ellie  /home
6  $ popd
   ~/home
   $ pwd
   /home/ellie
7  $ popd
   /home
   $ pwd
   /home
8  $ popd
   bash: popd: Directory stack empty.
```

说明

1. 第一个 pwd 命令显示当前工作目录，/home/ellie。下一个带..参数的 pushd 命令，将父目录(..)压入栈。pwd 的输出表明/home 在目录栈的顶部(从显示列的最左边开始)，用代字符号(~)表示的用户主目录在栈底。pushd 命令将转换到压入栈的那个目录；即，“..”被译成/home。新目录由第二个 pwd 命令显示出来。

2. 不带参数的 pushd 命令将栈顶的两个目录项互换且转换到交换后的目录。本例中，目录又转换回用户的主目录，/home/ellie。

3. pushd 命令将它的参数，~/perlclass，压入栈，并转换到相应目录。

4. 内置的 dirs 命令显示目录栈，栈顶的在显示列的最左边。代字符号表示用户的主目录。

5. 带-l 选项的 dirs 命令以完全路径格式而不是代字符号扩展格式列出目录栈。

6. popd 命令从栈顶删除一个目录，并转换到此目录。

7. popd 命令从栈顶删除另一个目录，并转换到此目录。

8. 因为栈已空，popd 命令不能删除任何目录项，bash 发出一个错误消息进行说明。

13.8 元字符(通配符)

元字符是一种可以用来代表自身以外的内容的特殊字符。shell 的元字符也被称作“通配符”。表 13-10 列出了 shell 的元字符及其功能。

表 13-10 元字符

元 字 符	含 义
\	按字面含义解释它后面那个字符
&	在后台处理的进程
;	分隔命令
$	替换变量
?	匹配单个字符
[abc]	匹配这组字符中的一个。例如，a、b 或者 c
[!abc]	匹配这组字符以外的某个字符。例如，除 a、b 或者 c 以外的字符
*	匹配零个或多个字符
(cmds)	在子 shell 中执行命令
{cmds}	在当前 shell 中执行命令

13.9 文件名替换(globbing)

计算命令行时，shell 会用元字符来缩写能够匹配某个特定字符组的文件名或路径名。表 13-11 中所列的文件名替换元字符将被展开为一组按字母顺序排列的文件名。将元字符展开为文件名的过程又被称作文件名替换或 globbing。如果没有文件名能够跟所用的元字符匹配，shell 就会把这个元字符作为一个字面字符。

表 13-11 shell 元字符与文件名替换

元 字 符	含 义
*	匹配零个或多个字符
?	匹配一个字符
[abc]	匹配 a、b、c 这组字符中的一个
[!abc]	匹配 a、b、c 这组字符以外的某个字符
{a,ile,ax}	匹配一个或一组字符
[a–z]	匹配在 a 至 z 这个范围内的某个字符
[!a~z]	匹配不在 a 至 z 这个范围内的某个字符
\	转义或禁用后面那个元字符



13.9.1 星号

星号是一个通配符，它匹配文件名中零个或多个任意字符。

范例 13-41

```
1   $ ls *
    abc abc1 abc122 abc123 abc2 file1 file1.bak file2 file2.bak none
    nonsense nobody nothing nowhere one
2   $ ls *.bak
    file1.bak file2.bak
3   $ echo a*
    ab abc1 abc122 abc123 abc2
```

说明

1. 星号被展开为当前工作目录下的所有文件的名称。所有的文件名都作为参数传给 ls 并在显示出来。

2. 匹配并列出所有以零个或多个字符开头、.bak 结尾的文件名。

3. 匹配所有以 a 开头、后跟零个或多个字符的文件名，并将它们作为参数传给 echo 命令。

13.9.2 问号

问号代表文件名中某一单个字符。当文件名中包含一个或多个问号时，shell 把问号替换为在文件名中匹配到的字符，以这种方式来完成文件名替换。

范例 13-42

```
1   $ ls
    abc  abc122  abc2  file1.bak  file2.bak  nonsense  nothing  one
    abc1  abc123  file1  file2  none  noone  nowhere
2   $ ls a?c?
    abc1 abc2
3   $ ls ??
    ls: ??: No such file or directory
4   $ echo  abc???
    abc122 abc123
5   $ echo ??
    ??
```

说明

1. 列出当前目录下的文件。

2. 匹配并列出以 a 开头，后跟一个字符，再跟字符 c 和一个字符的文件名。

3. 如果找到正好由两个字符组成的文件名，就列出来。因为当前目录下没有名字为两个字符的文件，所以这两个问号被解释为由两个字面字符“?”组成的文件名。如果没有找到这样的文件，就打印出一条错误信息。

4. 扩展以 abc 开头、后跟正好 3 个字符的文件名，并用 echo 命令显示。

5. 当前目录下没有名字正好是两个字符的文件。shell 找不到匹配，就把问号当成一个

字面上的问号。

13.9.3　方括号

括号用于匹配包含指定字符组或字符范围内某个字符的文件名。

范例 13-43

```
1  $ ls
   abc  abc122 abc2 file1.bak file2.bak nonsense nothing
   one  abc1 abc123 file1 file2 none noone nowhere
2  $ ls abc[123]
   abc1  abc2
3  $ ls abc[1-3]
   abc1  abc2
4  $ ls [a-z][a-z][a-z]
   abc one
5  $ ls [!f-z]???
   abc1  abc2
6  $ ls abc12[23]
   abc122 abc123
```

说明

1. 列出当前目录下所有文件。

2. 匹配所有包含 4 个字符的文件名，列出以 abc 开头，后跟 1、2 或 3 的文件名。只匹配方括号中这组字符中的任一个。

3. 匹配所有包含 4 个字符的文件名，列出以 abc 开头，后跟一个 1~3 之间数字的文件名。

4. 匹配所有包含 3 个字符的文件名，列出由 3 个小写字母组成的文件名。

5. 列出所有包含 4 个字符、第 1 个字符不是 f 至 z 之间的小写字母([!f-z])，后面是 3 个任意字符的文件名，这里？代表一个字符。

6. 列出文件名以 abc12 开头，后跟 2 或 3 的文件。

13.9.4　花括号

花括号用来匹配一组用逗号分隔的字符串中的任一个。左花括号之前的所有字符称为前文(preamble)，右花括号之后的所有字符称为后文(preamble)。前文和后文都是可选的。花括号中不能包含不加引号的空白符。

范例 13-44

```
1  $ ls
   a.c b.c abc ab3 ab4 ab5 file1 file2 file3 file4 file5 foo
   faa fumble
2  $ ls f{oo,aa,umble}
   foo faa fumble
3  $ ls a{.c,c,b[3-5]}
   a.c ab3 ab4 ab5
4  $ mkdir /usr/local/src/bash/{old,new,dist,bugs}
```



```
5   $ chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
6   $ echo fo{o, um}*
    fo{o, um}*
7   $ echo {mam,pap,ba}a
    mama papa baa
8   $ echo post{script,office,ure}
    postscript postoffice posture
```

说明

1. 列出当前目录下的所有文件。

2. 匹配文件名以 f 开头，后跟括号中任一字符串(oo、aa 或 umble)的文件。若在括号中加入了空格符，则会出现错误信息“Missing}”。

3. 匹配文件名以 a 开头，后跟括号中任一字符串(.c、c、b3、b4 或 b5)的文件。花括号中允许使用方括号。

4. 在/usr/local/src/bash 目录下创建 4 个新目录，它们分别为：old，new，dist 和 bugs。

5. root 用户权限将指派给/usr/ucb 目录下的 ex 和 edit 文件，以及/usr/lib 目录下的文件名以 ex 开头，后跟 1 个字符，1 个句点，1 个或多个任意字符的文件和名为 how_ex 的文件。

6. 只要括号中出现未加引号的空格就不对括号进行扩展。

7. 括号扩展不见得总是扩展文件名。本例中，括号后的字符 a 被添加到括号中的每个字符串后面并在结果中显示。

8. 前缀是一个字符串 post，后跟由括号括着的以逗号分隔的字符串。括号扩展已被执行并在结果中显示。

13.9.5 转义元字符

反斜杠用于屏蔽某一单个字符的特殊含义。被转义的字符将只代表其本身。

范例 13-45

```
1   $ ls
    abc file1 youx
2   $ echo How are you?
    How are youx
3   $ echo How are you\?
    How are you?
4   $ echo When does this line \
    > ever end\?
    When does this line ever end?
```

说明

1. 列出当前目录下的所有文件(注意 youx 文件)。

2. shell 对?执行文件名扩展。匹配当前目录下所有以 you 开头，后面有且只有一个字符的文件名，把它们替换到字符串中。文件名 youx 将被替换到字符串中，把字符串变成：How are youx (这好像不是我们想要的结果)。

3. 在问号前面加一个反斜杠，问号就被转义了，这意味着 shell 不会再把它当作通配

符来解释。

4. 在换行符前面加一个反斜杠将其转义。次提示符将一直出现直到字符串被换行符终止。转义问号(?)，不对它执行文件名替换。

13.9.6 代字符号和连字符扩展

代字符号被 bash shell(来自 C shell)用来作路径扩展。代字符号指代的是用户的主目录的完全路径名[11]。当给代字符号添加一个用户名时，它扩展成该用户的全路径名。

当在代字符号后跟一个加号时，PWD(当前工作目录)的值将替换代字符号。当代字符号后跟一个连字号时，将被上一个工作目录替换。OLDPWD 也是指上一个工作目录。

范例 13-46

```
1   $ echo ~
    /home/jody/ellie
2   $ echo ~joe
    /home/joe
3   $ echo ~+
    /home/jody/ellie/perl
4   $ echo ~-
    /home/jody/ellie/prac
5   $ echo $OLDPWD
    /home/jody/ellie/prac
6   $ cd -
    /home/jody/ellie/prac
```

说明

1. 代字符号的值为用户的主目录的完全路径名。
2. 在用户名前的代字符号的值为 joe 的主目录的完全路径名。
3. ~+符号的值为工作目录的完全路径名。
4. ~-符号的值为上一个工作目录。
5. OLDPWD 变量包含上一个工作目录。
6. 连字符代表上一个工作目录。cd 命令回到上一个工作目录并显示该目录。

13.9.7 控制通配符(globbing)

一旦设置了 bash 变量 noglob，或给 set 命令带了-f 选项，文件名替换(也称为 globbing)功能就被关闭，这意味着所有的元字符都将只代表其自身，它们不再被用作通配符。使用 grep、sed 或 awk 之类的程序搜索模式时，要用到关闭元字符的功能，因为 shell 可能会试图去展开这些程序使用的元字符。如果没有设置 globbing，所有的元字符必须用一个反斜杠来转义以关闭通配符解释。

内置的 shopt 命令(bash 2.x 版本)也支持控制 globbing 的选项。

---

[11] 代字符号如果出现在双引号或单引号中时将不会被扩展。



范例 13-47

```
1   $ set noglob or set -f
2   $ print * ?? [] ~ $LOGNAME
    * ?? [] /home/jody/ellie ellie
3   $ unset noglob or set +f
4   $ shopt -s dotglob   # Only available in bash versions 2.x
5   $ echo *bash*
    .bash_history .bash_logout .bash_profile .bashrc bashnote
    bashtest
```

说明

1. –f 选项作为 set 命令的参数。它关闭通配符用于文件名扩展的特殊含义。

2. 文件名扩展元字符被显示为其自身，没有经过任何解释。注意代字符号和美元符仍然被扩展了，因为它们没有用于文件名扩展。

3. 如果没有设置 noglob 或设置了+f 选项，文件名元字符会被扩展。

4. shopt 内置命令可以为 shell 设置选项。dotglob 选项允许文件名用 globbing 元字符匹配，甚至是以一个点开头的文件名。通常以点开头的文件是不可见的，并且在执行文件名扩展时不会被识别。

5. 因为在第 4 行设置了 dotglob 选项，所以当通配符*用于文件名扩展时，如果以点开头的文件名包含模式 bash，它也会被扩展。

13.9.8 扩展的文件名 globbing(bash 2.x)

bash 2.x 还包含这种来自 Korn shell 的模式匹配功能，允许正则表达式类型的语法(参见表 13-12)。除非 shopt 命令的 extglob 选项被打开：

```
shopt -s extglob
```

否则正则表达式操作符不会被识别。

表 13-12 扩展的模式匹配

正则表达式	含 义
abc?(2|9)1	?与零个或一个出现的括号中的任意模式相匹配。竖杠代表一个或条件。例如，2 或 9。匹配 abc21、abc91 或者 abc1
abc*([0-9])	*与零个或多个出现的括号中的任意模式相匹配。匹配 abc 后跟零个或多个数字。如，abc、abc1234、abc3、abc2 等
abc+([0-9])	+与一个或多个出现的括号中的任意模式相匹配。匹配 abc 后跟一个或多个数字。例如，abc3、abc123 等
no@(one|ne)	@正好与括号中任一模式相匹配。匹配 noone 或 none
no!(thing|where)	!与括弧中的任意模式除外的其他所有串相匹配。匹配 no、nobody 或 noone，但不能是 nothing 或 nowhere

范例 13-48

```
1  $ shopt -s extglob
2  $ ls
   abc      abc122   f1   f3      nonsense   nothing   one
   abc1     abc2     f2   none    noone      nowhere
3  $ ls abc?(1|2)
   abc     abc1     abc2
4  $ ls abc*([1-5])
   abc     abc1     abc122     abc2
5  $ ls abc+([0-5])
   abc1    abc122    abc2
6  $ ls no@(thing|ne)
   none    nothing
7  $ ls no!(thing)
   none    nonsense   noone     nowhere
```

说明

1. shopt 内置命令用来设置 extglob(扩展 globbing)选项，允许 bash 识别扩展的模式匹配字符。

2. 列出当前工作目录下的所有文件。

3. 匹配以 abc 开头后跟零个或一个括弧中的任意模式的文件名。匹配 abc、abc1 或 abc2。

4. 匹配以 abc 开头后跟零个或多个 1~5 之间的数字的文件名。匹配 abc、abc1、abc122、abc123 和 abc2。

5. 匹配以 abc 开头后跟一个或多个 0~5 之间的数字的文件名。匹配 abc1、abc122、abc123 和 abc2。

6. 匹配以 no 开头后跟指定串 thing 或 ne 的文件名。匹配 nothing 或 none。

7. 匹配以 no 开头后跟除 thing 外的字符串的文件名。匹配 none、nonsense、noone 和 nowhere。!意思是非。

13.10 变量

13.10.1 变量类型

变量可分为两类：局部变量和环境变量。局部变量只在创建它们的 shell 中可用。而环境变量则可以在创建它们的 shell 及其派生出来的任意子进程中使用。有些变量是用户创建的，其他的则是专用 shell 变量。

13.10.2 命名惯例

变量名必须以字母或下划线字符开头。其余的字符可以是字母、数字(0~9)或下划线字符。任何其他的字符都标志着变量名的终止。名字是大小写敏感的。给变量赋值时，等号周围不能有任何空白符。为了给变量赋空值，可以在等号后跟一个换行符。创建一个局部变量最简单的格式是给一个变量赋值，如以下格式所示。



格式

变量=值

范例 13-49

```
name=Tommy
```

13.10.3 内置命令 declare

有两个内置命令可以用来创建变量，它们是 declare 和 typeset，其选项可以控制变量设置的方式。typeset 命令(来自 Korn shell)的功能和 declare 命令(bash)完全一样。bash 文档中指出，“提供 typeset 命令是为了和 Korn shell 兼容。但是不建议使用它，而应使用内置命令 declare”[12]。因此从这一点来说，我们将使用 declare 命令(即使我们使用 typeset 也一样方便)。

不带任何参数时，declare 将列出所有已设置的变量。通常只读变量不能被重新赋值或复位。如果只读变量是用 declare 创建的，那它们不可以被复位，但可以被重新赋值。整型变量也可以用 declare 赋值。

格式

declare 变量=值

范例 13-50

```
declare name=Tommy
```

表 13-13 declare 选项

选 项	含 义
-a[13]	将变量当作一个数组。即，分配元素
-f	列出函数的名称和定义
-F[13]	只列出函数名
-i	将变量设为整型
-r	将变量设为只读
-x	将变量名输出到子 shell 中

13.10.4 局部变量和作用域

变量的作用域指变量在一个程序中的哪些地方可见。对于 shell 来说，局部变量的作用域被限定在创建它们的 shell 中。

给变量赋值时，等号前后不能有空白符。如果要给变量赋空值，可以在等号后跟一个换行符[14]。

---

⑫ bash 参考手册：http://www.delorie.com/gnu/docs/bash/bashref_56.html。

⑬ -a 和-F 只在 bash 2.x 版中应用。

⑭ 运行 set 命令时，凡是被设置为某个值或空值的变量都会被显示出来，未经设置的变量则不会被显示。

变量前的美元符用来提取存储在变量里的值。

local 函数可以用来创建局部变量，但仅限于在函数内使用(请参见 13.16.1 节，“定义函数”)。

设置局部变量　局部变量可以通过简单地赋予它一个值或一个变量名来设置，或者如范例 13-51 所示用 declare 内置函数来设置。

范例 13-51

```
1   $ round=world or declare round=world
    $ echo $round
    world
2   $ name="Peter Piper"
    $ echo $name
    Peter Piper
3   $ x=
    $ echo $x
4   $ file.bak="$HOME/junk"
    bash: file.bak=/home/jody/ellie/junk: not found
```

说明

1. 将局部变量 round 赋值为 world。如果遇到变量名前面是个美元符的情况，shell 就执行变量替换。该命令显示变量 round 的值(不要混淆提示符($)和用来执行变量替换的$符)。

2. 局部变量 name 被赋值为“Peter Piper”。必须用引号来保护空白符，这样，shell 分析命令行时才不会将这个字符串看成是两个词。该命令显示变量 name 的值。

3. 这条命令没有给变量 x 赋值，因此 x 被赋值为空。结果显示一个空值，即空字符串。

4. 变量名中出现句点是非法的。变量名可以使用的字符只能是数字、字母和下划线。因此，shell 尝试将这个字符串作为一条命令来执行。

范例 13-52

```
1   $ echo $$
    1313
2   $ round=world
    $ echo $round
    world
3   $ bash          # Start a subshell
4   $ echo $$
    1326
5   $ echo $round

6   $ exit          # Exits this shell, returns to parent shell
7   $ echo $$
    1313
8   $ echo $round
    world
```

说明

1. 双美元符变量的值是当前 shell 的 PID。本例中这个 shell 的 PID 是 1313。



2. 将局部变量 round 赋值为字符串 world，并打印该变量的值。

3. 另外启动一个 bash shell，这个 shell 被称为子 shell(subshell 或 child shell)。

4. 当前这个 shell 的 PID 时 1326，其父 shell 的 PID 则是 1313。

5. 局部变量 round 在这个 shell 中没有定义，因此打印了一个空行。

6. exit 命令终止当前 shell 并返回父 shell(也可以按 Ctrl+D 组合键退出当前 shell)。

7. 返回到父 shell。显示它的 PID。

8. 显示变量 round 的值。它是这个 shell 的局部变量。

设置只读变量 只读变量是不能被重新定义或复位的特殊变量。但是，如果使用了 declare 函数，只读变量可以被重新定义，但不能被复位。

范例 13-53

```
1   $ name=Tom
2   $ readonly name
    $ echo $name
    Tom
3   $ unset name
    bash: unset: name: cannot unset: readonly variable
4   $ name=Joe
    bash: name: readonly variable
5   $ declare -r city='Santa Clara'
6   $ unset city
    bash: unset: city: cannot unset: readonly variable
7   $ declare city='San Francisco'   # What happened here?
    $ echo $city
    San Francisco
```

说明

1. 局部变量 name 的值被设为 Tom。

2. 将变量 name 设为只读。

3. 不能复位只读变量。

4. 不能重新定义只读变量。

5. declare 内置命令给只读变量 city 赋值 Santa Clara。当所赋值串中包含空白符时必须用引号。

6. 因其是只读变量，所以不能被复位。

7. 当只读变量是由 declare 命令创建时，它不可以被复位，但可以被重新赋值。

13.10.5 环境变量

环境变量可用在创建它们的 shell 和从该 shell 派生的任意子 shell 或进程中。它们通常被称为全局变量，以区别于局部变量。通常，环境变量应该大写。环境变量是已经用 export 内置命令导出的变量。

变量被创建时所处的 shell 被称为父 shell。如果父 shell 又启动了一个 shell，这个新的 shell 被称作子 shell。环境变量将传递给从创建它们的 shell 里启动的任意子进程。它们从父亲传递给儿子再到孙子等，但是不可向其他方向传递。比如，一个子进程可以创建环境

变量，但不能将它传回给它的父进程，只能传给它的子进程[15]。有一些环境变量，比如 HOME、LOGNAME、PATH 和 SHELL，在用户登录之前就已经被/bin/login 程序设置好了。通常，环境变量定义并保存在用户主目录下的.bash_profile 文件中。请参见表 13-14 中列出的环境变量。

表 13-14 bash 环境变量

变量名	含义
_(下划线)	上一条命令的最后一个参数
BASH	展开为调用 bash 实例时使用的全路径名
BASH_ENV	和 ENV 一样，但只可在 bash 2.0 或更高版本中设置[16]
BASH_VERSINFO	使用 2.0 以上版本的 bash 时，展开为版本信息[16]
BASH_VERSION	展开为当前 bash 实例的版本号
CDPATH	cd 命令的搜索路径。它是以冒号分隔的目录列表，shell 通过它来搜索 cd 命令指定的目标目录。例如.:~:/usr
COLUMNS	设置该变量就给 shell 编辑模式和选择的命令定义了编辑窗口的宽度
DIRSTACK	在 2.0 或以上版本的 bash 中，代表目录栈的当前内容[16]
EDITOR	内置编辑器 emacs、gmacs 或 vi 的路径名
ENV	每一个新的 bash shell(包括脚本)启动时执行的环境文件。通常赋予这个变量的文件名是.bashrc。ENV 的值被解释为路径名前，shell 先要对其进行参量扩展，命令替换和算术扩展
EUID	展开为在 shell 启动时被初始化的当前用户的有效 ID
FCEDIT	fc 命令的默认编辑器名
FIGNORE	执行文件名补全时可忽略的以冒号分隔的后缀列表。以 FIGNORE 中任一项为后缀的文件名被从匹配的文件名列表中排除。例如值为.o:~
FORMAT	用来格式化在命令管道上的 time 关键字的输出
GLOBIGNORE	在文件名扩展(称为 globbing)时被忽略的文件列表[16]
GROUPS	当前用户所属的组[16]
HISTCMD	当前命令的历史编号或在历史清单中的序号。如果 HISTCMD 被复位，即使它随后就会重置，也将失去它的特殊属性
HISTCONTROL	如果设置了 ignorespace 值，以一个空格符开头的行将不会进入历史清单。如果设置了 ignoredups 值，那和前一个历史行匹配的行不会进入。值 ignoreboth 结合了这两个选项。如果被复位，或设置成除了上面所说的任意其他值时，所有被解释器所读的行都将保存到历史清单中
HISTFILE	指定保存命令行历史的文件。默认值是~/.bash_history。如果被复位，交互式 shell 退出时将不保存命令行历史

⑮ 就像 DNA，只能按从父亲到儿子这个方向遗传。

⑯ 在 bash 2.x 以前版本中没有。



(续表)

变 量 名	含 义
HISTFILESIZE	历史文件能包含的最大行数。当给这个变量赋值后，如果有必要，历史文件将被截尾，以使包含的行数不超过这个数。默认值是 500
HISTSIZE	记录在命令行历史文件中的命令数。默认是 500
HOME	主目录。未指定目录时，cd 命令将转向该目录
HOSTFILE	包含一个格式和/etc/hosts 一样的文件的名称，当 shell 需要补全一个主机名时将读取该文件。文件可以交互式更改。下一次试图补全主机名时，bash 将新文件的内容添加到已经存在的数据库中
HOSTTYPE	自动设置正在运行 bash 的机器的类型。默认值是由系统决定的
IFS	内部字段分隔符，一般是空格符、制表符和换行符，用于由命令替换，循环结构中的表和读取的输入产生的词的字段划分
IGNOREEOF	控制 shell 接收到单独一个 EOF 字符作为输入时的行为。如果设置，它的值就是shell退出前在一个输入行的最前面键入的连续EOF字符的个数。如果变量存在但没有一个数字值，或没有值，那么默认值是 10。如果它不存在，EOF 意味着给 shell 的输入的终止。它只在交互式 shell 中有效
INPUTRC	readline 启动文件的文件名，取代默认的~/.inputrc
LANG	用来为没有以 LC_开头的变量明确选取的种类确定 locale 类
LC_ALL	忽略 LANG 和任何其他 LC_变量的值[16]
LC_COLLATE	确定对路径名扩展的结果进行排序时的整理顺序，以及匹配文件名与模式时的范围表达式，等价类和整理序列的行为[16]
LC_MESSAGES	确定用于转换前面有一个$的双引号串的 locale[16]
LINENO	每次 shell 在一个脚本或函数中替换代表当前连续行号(从 1 开始)的十进制数时，都将引用该参数[16]
MACHTYPE	包含一个描述正在运行 bash 的系统的串
MAIL	如果该参数被设置为某个邮件文件的名称，而 MAILPATH 未被设置，当邮件到达 MAIL 指定的文件时，shell 会通知用户
MAIL_WARNING	如果设置了该变量，当 bash 发现用于检查邮件的文件在上次检查后又被访问了，将打印消息“The mail in [filename where mail is stored] has been read”
MAILCHECK	这个参数定义 shell 将隔多长时间(以秒为单位)检查一次由参数 MAILPATH 或 MAILFILE 指定的文件，看看是否有邮件到达。默认值是 600 秒(10 分钟)。如果将它设为 0，shell 每次输出主提示符之前都会去检查邮件
MAILPATH	由冒号分隔的文件名列表。如果设置了这个参数，只要有邮件到达任何一个由它指定的文件，shell 都会通知用户。每个文件名后面都可以跟一个百分号和一条消息，当文件修改时间发生变化时，shell 会显示这条消息。默认的消息是：You have mail

(续表)

变 量 名	含 义
OLDPWD	前一个工作目录
OPTARG	上一个由 getopts 内置命令处理的选项参数的值
OPTERR	如果设置成 1，显示来自 getopts 内置命令的错误信息
OPTIND	下一个由 getopts 内置命令处理的参数的序号
OSTYPE	自动设置成一个串，该串描述正在运行 bash 的操作系统。默认值由系统决定
PATH	命令搜索路径。一个由冒号分隔的目录列表，shell 用它来搜索命令。默认路径由系统决定，并且由安装 bash 的管理员设置。一个普通值为 /usr/gnu/bin:/usr/local/bin:/usr/ucb:/usr/bin:
PIPESTATUS	一个数组，包含一列最近在管道执行的前台作业的进程退出状态值
PPID	父进程的进程 ID
PROMPT_COMMAND	赋给这个变量的命令将在主提示符显示前执行
PS1	主提示符串，默认值是$
PS2	次提示符串，默认值是>
PS3	与 select 命令一起使用的选择提示符串，默认值是#?
PS4	当开启追踪时使用的调试提示符串，默认值是+。追踪可以用 set –x 开启
PWD	当前工作目录。由 cd 设置
RANDOM	每次引用该变量，就产生一个随机整数。随机数序列可以通过给 RANDOM 赋值来初始化。如果 RANDOM 被复位，即使随后再设置，它也将失去特定的属性
REPLY	当没有给 read 提供参数时设置
SECONDS	每次 SECONDS 被引用，将返回调用 shell 以来的秒数。如果给 SECONDS 赋一个值，以后引用返回的值将是赋值以来的秒数加上所赋的值。如果 SECONDS 被复位，即使随后再设置，它也将失去特定的属性
SHELL	当调用 shell 时，它扫描环境变量以寻找该名字。shell 给 PATH、PS1、PS2、MAILCHECK 和 IFS 设置默认值。HOME 和 MAIL 由 login(1)设置
SHELLOPTS	包含一列开启的 shell 选项，比如 braceexpand、hashall、monitor 等
SHLVL	每启动一个 bash 实例时将其加 1
TMOUT	设置退出前等待输入的秒数
UID	展开为当前用户的用户 ID，在 shell 启动时初始化

设置环境变量 如果想设置环境变量，就要在给变量赋值之后或设置变量时使用 export 命令(参见表 13-15)。带-x 选项的 declare 内置命令也可完成同样的功能(输出变量时不要在变量名前面加$)。



表 13-15　export 命令和它的选项

选　项	值
--	标志着选项末尾，余下的参数被视为变量
-f	名-值对被看作函数，而不是变量
-n	将一个全局(导出)变量转换成局部变量。之后该变量将不能被导出到子进程中
-p	显示所有的全局变量

格式

```
export 变量=值
变量=值； export 变量
declare -x 变量=值
```

范例 13-54

```
export NAME=john
PS1= '\d:\W:$USER> ' ; export PS1
declare -x TERM=sun
```

范例 13-55

```
1   $ export TERM=sun     # or declare -x TERM=sun
2   $ NAME="John Smith"
    $ export NAME
    $ echo $NAME
    John Smith
3   $ echo $$
    319             # pid number for parent shell
4   $ bash          # Start a subshell
5   $ echo $$
    340             # pid number for new shell
6   $ echo $NAME
    John Smith
7   $ declare -x NAME="April Jenner"
    $ echo $NAME
    April Jenner
8   $ exit           # Exit the subshell and go back to parent shell
9   $ echo $$
    319             # pid number for parent shell
10  $ echo $NAME
    John Smith
```

说明

1. 给 TERM 变量赋值 sun。同时输出它。现在，由这个 shell 启动的所有进程都将继承这个变量。也可以用 declare －x 来完成同样的功能。

2. 定义并输出变量 NAME，让它可以被当前 shell 启动的所有子 shell 使用。

3. 打印当前 shell 的 PID 的值。

4. 启动一个新的 bash。这个新 shell 称为子 shell。原来那个 shell 称为父 shell。

5. 新的 bash shell 的 PID 保存在变量$$中，回显这个变量的值。

6. 在父 shell 中设置的变量 NAME 导出给这个新 shell，这条命令显示它的值。

7. 内置的 declare 函数是设置变量的另一种方式。用-x 开关，declare 可以输出变量。该变量被重新设置为 April Jenner。这个变化将输出到所有的子 shell，但不会影响父 shell。输出的变量不会向上传递给父 shell。

8. 退出这个 bash 子 shell。

9. 再次显示父 shell 的 PID。

10. 变量 NANE 的值还跟原来一样。从父 shell 输出到子 shell 时，变量保持它们的值不变。子 shell 不可能改变父 shell 的变量的值。

13.10.6 复位变量

只要不被设为只读，局部变量和环境变量都可以用 unset 命令复位。

范例 13-56

```
unset name; unset TERM
```

说明

unset 命令从 shell 存储器中删除变量。

13.10.7 显示变量值

echo 命令 内置 echo 命令将它的参数显示到标准输出上。echo 加-e 选项，允许使用大量控制输出外观的转义序列。表 13-16 列出了 echo 选项和转义序列。

表 13-16 echo 选项和转义序列

选 项	含 义
-e	允许解释下面列出的转义序列
-E	禁止解释这些转义字符，即使在那些默认解释它们的系统上(bash 2.x)[17]
-n	删除输出结果中行尾的换行符
转义序列	
\a	报警(铃)[17]
\b	退格
\c	不带换行符打印一行
\f	换页
\n	换行
\r	回车
\t	制表符
\v	纵向制表符
\\	反斜杠
\nnn	ASCII 码是 nnn(八进制)的字符

⑰ 在 2.0 以前版本的 bash 不提供该功能。

当使用转义序列时，不要忘记用-e 开头。

范例 13-57

```
1   $ echo The username is $LOGNAME.
    The username is ellie.
2   $ echo -e "\t\tHello there\c"
                Hello there$
3   $ echo -n "Hello there"
    Hello there$
```

说明

1. echo 命令将它的参数打印到屏幕上。shell 会在执行 echo 命令之前先进行变量替换。
2. 带-e 选项的 echo 命令支持转义序列，和 C 编程语言类似。$是 shell 提示符。
3. 当-n 选项打开时，不带换行符打印一行。这个版本的 echo 不支持转义序列。

printf 命令 printf[18]的 GNU 版本可以用来编排打印输出的格式。它以和 C printf 函数相同的方式打印格式串。格式由一个串组成，它包含描述打印输出结果的格式指令。格式指令由带格式符(diouxXfeEgGcs)的%指定，%f 代表一个浮点数，%d 则代表一个(十进制)整数。

要得到 printf 格式符的完整清单以及如何使用它们，可以在命令行键入：printf --help。键入 printf --version 就可以知道使用的 printf 是什么版本。如果使用的是 bash 2.x，内置 printf 命令所用的格式和/usr/bin 下的 printf 可执行程序完全一样。

格式

```
printf 格式[参数...]
```

范例 13-58

```
printf "%10.2f%5d\n" 10.5  25
```

表 13-17 printf 命令的格式符

格 式 符	值
\"	双引号
\0NNN	一个八进制字符，这里 NNN 代表 0~3 位
\\	反斜杠
\a	报警或蜂鸣
\b	退格
\c	不产生更多的输出
\f	换页
\n	换行
\r	回车

⑱ 在 bash 2.x 版本中，printf 是一个内置命令。

(续表)

格式说明	值
\t	水平制表符
\v	垂直制表符
\xNNN	十六进制字符，这里 NNN 是 1~3 位
%%	单个百分号
%b	字符串参数，也对\转义字符进行解释

范例 13-59

```
1  $ printf --version
   printf (GNU sh-utils) 1.16
2  $ type printf
   printf is a shell builtin
3  $ printf "The number is %.2f\n" 100
   The number is 100.00
4  $ printf "%-20s%-15s%10.2f\n" "Jody" "Savage" 28
   Jody                Savage              28.00
5  $ printf "|%-20s|%-15s|%10.2f|\n" "Jody" "Savage" 28
   Jody                |Savage         |     28.00|
6  $ printf "%s's average was %.1f%%.\n" "Jody" $(( (80+70+90)/3 ))
   Jody's average was 80.0%.
```

说明

1. 显示 printf 命令的 GNU 版本。

2. 如果使用的是 bash 2.x，printf 是一个内置命令。

3. 按照说明符%.2f 指定的格式，参数 100 以保留两位小数的浮点数形式输出。与 C 函数不同的是，这里不需要用逗号来分隔参数。

4. 格式串指明将进行 3 个变换：第一个是%-20s(一个左对齐，长度为 20 的字符串)，接着是%-15s(一个左对齐，长度为 15 的字符串)，最后一个则是%10.2f(一个右对齐，长度为 10 的浮点数，其中的一个字符是句点，最后两个字符是小数点右边的两个数)。参数按对应%号的顺序被格式化，因此字符串 Jody 对应第一个%，字符串 Savage 对应第二个%，数字 28 则对应最后一个%号。

5. 该行和第 4 行一样，唯一的区别是增加了竖杠以说明串是左对齐还是右对齐的。

6. printf 命令格式化字符串 Jody 和算术扩展的结果(请参见 13.13 节，“算术扩展”)。需要两个百分号(%%)才能输出一个百分号(%)。

13.10.8 变量扩展修饰符

我们可以用一些专用修饰符来测试和修改变量。修饰符首先提供一个简单的条件测试，用来检查某个变量是否已经被设置，然后根据测试结果给变量赋一个值。请参见表 13-18 列出的变量修饰符。



表 13-18　变量修饰符

修　饰　符	值
${variable:-word}	如果变量 variable 已被设置且非空，则代入它的值。否则，代入 word
${variable:=word}	已被设置且值非空，就代入它的值。否则，将 variable 的值设为 word。始终代入 variable 的值。位置参量不能用这种方式赋值
${variable:+word}	如果变量 variable 已被设置且值非空，代入 word。否则，什么都不代入(代入空值)
${variable:?word}	如果变量 variable 已被设置且值非空，就代入它的值。否则，输出 word 并且从 shell 退出。如果省略了 word，就会显示信息：parameter null or not set
${variable:offset}	获得变量 variable 值中位置从 offset 开始的子串，偏移为从 0 到串的末尾[19]
${variable:offset:length}	获得变量 variable 值中位置从 offset 开始长度为 length 的子串

和冒号配合使用时，修饰符(-、=、+、?)检查变量是否尚未赋值或值为空。不加冒号时，值为空的变量也被认为已设置。

范例 13-60

(临时替换默认值)

```
1   $ fruit=peach
2   $ echo ${fruit:-plum}
    peach
3   $ echo ${newfruit:-apple}
    apple
4   $ echo $newfruit
5   $ echo $EDITOR       # More realistic example
6   $ echo ${EDITOR:-/bin/vi}
    /bin/vi
7   $ echo $EDITOR
8   $ name=
    $ echo ${name-Joe}
9   $ echo ${name:-Joe}
    Joe
```

说明

1. 将变量 fruit 的值设为 peach。

2. 这个专用修饰符将检查变量 fruit 是否已被设置。如果 fruit 已被设置，就显示它。否则，用 plum 替换 fruit，并显示该值。

3. 变量 newfruit 未曾被设置。值 apple 将暂时替换 newfruit。

4. 上一行的设置是暂时的，因此，变量 newfruit 仍未被设置。

5. 环境变量 EDITOR 尚未被设置。

6. 修饰符:-将/bin/vi 替换为 EDITOR。

---

⑲ 2.0 以前版本的 bash 不提供该功能。

7. EDITOR 未曾被设置过，因此什么都不会打印。

8. 变量 name 被设为空值。因为修饰符前面没有冒号，变量即使为空也被认为是设置过的，所以没有把新的值 Joe 赋给变量 name。

9. 冒号使得修饰符检查变量是否未设置或为空。只要是这两种情况之一，就用值 Joe 替换 name。

范例 13-61
（永久替换默认值）

```
1   $ name=
2   $ echo  ${name:=Peter}
    Peter
3   $ echo $name
    Peter
4   $ echo ${EDITOR:=/bin/vi}
    /bin/vi
5   $ echo $EDITOR
    /bin/vi
```

说明

1. 赋给变量 name 一个空值。

2. 用修饰符:=将检查变量 name 是否尚未被设置。如果已经被设置过了，就不会被改变。如果尚未设置或值为空，就将等号右边的值赋给它。由于之前已将变量 name 设置为空，所以现在要把 Peter 赋给它。这个设置是持久的。

3. 变量 name 的值还是 Peter。

4. 把变量 EDITOR 设置为/bin/vi。

5. 显示变量 EDITOR 的值。

范例 13-62
（临时替换值）

```
1   $ foo=grapes
2   $ echo ${foo:+pears}
    pears
3   $ echo $foo
    grapes
    $
```

说明

1. 将变量 foo 的值设置为 grapes。

2. 专用修饰符:=将检查变量 name 是否已被设置。如果已经被设置过，就用 pears 暂时替换 foo。否则，返回空。

3. 变量 foo 的值还是原来的值。

范例 13-63
（基于默认值创建错误信息）

```
1   $ echo ${namex:?"namex is undefined"}
```



```
    namex: namex is undefined
2   $ echo ${y?}
    y: parameter null or not set
```

说明

1. 修饰赋:?检查变量是否已被设置。如果尚未设置该变量，就把问号右边的信息打印在标准错误输出上。如果此时是在执行脚本，就退出脚本。

2. 如果问号后面没有提供报错信息，shell 就向标准错误输出发送默认的消息。

范例 13-64
(创建子串[21])

```
1   $ var=notebook
2   $ echo ${var:0:4}
    note
3   $ echo ${var:4:4}
    book
4   $ echo ${var:0:2}
    no
```

说明

1. 给变量赋值 notebook。

2. var 的子串从偏移 0(notebook 中的 n)开始，长度为 4 个字符，在 e 处结束。

3. var 的子串从偏移 4(notebook 中的 b)开始，长度为 4 个字符，在 k 处结束。

4. var 的子串从偏移 0(notebook 中的 n)开始，长度为 2 个字符，在 o 处结束。

13.10.9 子串的变量扩展

模式匹配变量用来在串首或串尾截掉串的某一特定部分。这些操作符最常见的用法是从路径头或尾删除路径名元素。如表 13-19 所示。

表 13-19 变量扩展子串[21]

表 达 式	功 能
${变量%模式}	将变量值的尾部与模式进行最小匹配，并将匹配到的部分删除
${变量%%模式}	将变量值的尾部与模式进行最大匹配，并将匹配到的部分删除
${变量#模式}	将变量值的头部与模式进行最小匹配，并将匹配到的部分删除
${变量##模式}	将变量值的头部与模式进行最大匹配，并将匹配到的部分删除
${#变量}	替换为变量中的字符个数。如果是*或@，长度则是位置参量的个数

范例 13-65

```
1   $ pathname="/usr/bin/local/bin"
2   $ echo ${pathname%/bin*}
    /usr/bin/local
```

---

[21] 2.x 以前版本的 bash 不提供该功能。

说明

1. 给局部变量 pathname 赋值/usr/bin/local/bin。
2. %删除路径名尾部包含模式/bin，后跟零个或多个字符的最小部分。即删除/bin。

范例 13-66

```
1   $ pathname="usr/bin/local/bin"
2   $ echo ${pathname%%/bin*}
    /usr
```

说明

1. 给局部变量 pathname 赋值/usr/bin/local/bin。
2. %%删除路径名尾部包含模式/bin，后跟零个或多个字符的最大部分。即删除/bin/local/bin。

范例 13-67

```
1   $ pathname=/home/lilliput/jake/.bashrc
2   $ echo ${pathname#/home}
    /lilliput/jake/.bashrc
```

说明

1. 给局部变量 pathname 赋值/home/liliput/jake/.bashrc。
2. #删除路径名头部包含模式/home 的最小部分。路径变量开头的/home 被删除。

范例 13-68

```
1   $ pathname=/home/lilliput/jake/.bashrc
2   $ echo ${pathname##*/}
    .bashrc
```

说明

1. 给局部变量 pathname 赋值/home/liliput/jake/.bashrc。
2. ##删除路径名的头部包含零个或多个字符，直到并包括最后一个斜杠的最大部分。即从路径变量中删除/home/liliput/jake/。

范例 13-69

```
1   $ name="Ebenezer Scrooge"
2   $ echo  ${#name}
    16
```

说明

1. 给变量 name 赋值 Ebenezer Scrooge。
2. ${#variable}语法显示赋给变量 name 的字符串中字符的个数。字符串 Ebenezer Scrooge 中有 16 个字符。



13.10.10 位置参量

这组专用内置变量常常被称为位置参量，通常被 shell 脚本用来从命令行接收参数，或者被函数用来保存传给它的参数。这组变量之所以被称为位置参量，是因为引用它们要用到 1、2、3 等数字，这些数字分别代表它们在参数列表中的相应位置。请参见表 13-20。

表 13-20 位置参量

表达式	功能
$0	指代当前 shell 脚本的名称
$1-$9	代表第 1 个到第 9 个位置参量
${10}	第 10 个位置参量
$#	其值为位置参量的个数
$*	其值为所有的位置参量
$@	除了被双引号引用的情况，含义与$*相同
"$*"	其值为"$1 $2 $3"
"$@"	其值为"$1" "$2" "$3"

shell 脚本名存在变量$0 中。位置参量可以用 set 命令来设置，重置和复位。

范例 13-70

```
1   $ set punky tommy bert jody
    $ echo $*          # Prints all the positional parameters
    punky tommy bert jody
2   $ echo $1          # Prints the first position
    punky
3   $ echo $2 $3       # Prints the second and third position
    tommy bert
4   $ echo $#          # Prints the total number of positional parameters
    4
5   $ set a b c d e f g h i j k l m
    $ print $10        # Prints the first positional parameter followed by a 0
    a0
    $ echo ${10} ${11} # Prints the 10th and 11th positions
    j k
6   $ echo $#
    13
7   $ echo $*
    a b c d e f g h i j k l m
8   $ set file1 file2 file3
    $ echo \$$#
    $3
9   $ eval echo \$$#
    file3
10  $ set --            # Unsets all positional parameters
```

说明

1. set 命令给位置参量赋值。专用变量$*包含所有的位置参量。
2. 显示第 1 个位置参量的值，punky。
3. 显示第 2 和第 3 个位置参量的值，tommy 和 bert。
4. 专用变量$#的值是当前已设置的位置参量的个数。
5. set 命令复位所有的位置参量。原来的位置参量集被清除。要打印 9 以上的任意位置参量，就要用花括号把两个数字括起来。否则，就打印第一个位置参量的值，后跟另一个数。
6. 位置参量个数现在是 13。
7. 显示所有位置参量的值。
8. 美元符被转义，$#是参数个数。echo 命令显示$3，一个美元符号后跟位置参量的个数。
9. 执行命令之前，eval 命令对命令行进行第二次解析。第一次由 shell 解析，将输出$3。第二次由 eval 解析，显示$3 的值，即 file3。
10. 带--选项的 set 命令清除或复位所有的位置参量。

13.10.11 其他特殊变量

shell 有一些由单个字符组成的特殊变量。在字符前面加上美元符就能访问变量中保存的值。如表 13-21 所示。

表 13-21 特殊变量

变　量	含　义
$	当前 shell 的 PID
-	当前的 sh 选项设置
?	已执行的上一条命令的退出值
!	最后一个进入后台的作业的 PID

范例 13-71

```
1   $ echo The pid of this shell is $$
    The pid of this shell is 4725
2   $ echo The options for this shell are $-
    The options for this shell are imh
3   $ grep dodo /etc/passwd
    $ echo $?
    1
4   $ sleep 25&
    4736
    $ echo $!
    4736
```

说明

1. 变量$保存这个进程的 PID 值。
2. 变量 - 列出当前这个交互式 bash shell 的所有选项。



3. grep 命令在/etc/passwd 文件中查找字符串 dodo。变量? 保存了上一条被执行的命令的退出状态。由于 grep 返回的值是 1，因此可以假定 grep 的查找失败了。退出状态 0 代表成功退出。

4. 变量!保存上一条被放入后台的命令的 PID 号。sleep 命令后面的&把命令发到后台。

13.11 引用

引用被用来保护特殊的元字符不被解释和禁止参量扩展。引用有 3 种方式：反斜杠、单引号和双引号。表 13-22 列出的字符对 shell 而言都是特殊的，必须加引号。

表 13-22 需要引用的特殊元字符

元 字 符	含 义
;	命令分隔符
&	后台处理
()	命令编组。创建子 shell
{ }	命令编组。不创建子 shell
|	管道
<	输入重定向
>	输出重定向
newline	命令终止
space/tab	词分隔符
$	变量替换字符
* [] ?	用于文件名扩展的 shell 元字符

单引号和双引号都必须成对出现。单引号保护特殊元字符(如$、*、?、|、>和<)免受解释。双引号也能保护特殊元字符不受解释，但它允许处理变量替换字符(美元符)和命令替换字符(反引号)。单引号可以保护双引号，双引号也会保护单引号。

和 Bourne shell 不一样，如果有不匹配的引号，bash 会设法通知你。在交互式运行状态下，如果引号不匹配，就会出现次提示符。运行 shell 脚本时，shell 先扫描文件，如果有引号不匹配，shell 会试图找到下一个引号来匹配它。如果这种尝试失败，程序就会异常终止，终端上将出现这样一条信息："bash:unexpected EOF while looking for `"`"。即使是顶尖的 shell 程序员高手，也会遇到引用导致的问题。关于 shell 引用规则，请参见附录 C。

13.11.1 反斜杠

反斜杠用于引用(或转义)单个字符，使其免受解释。单引号里的反斜杠不会被解释。如果是在双括号里，反斜杠将保护美元符($)、反引号(``)和反斜杠不被解释。

范例 13-72

```
1   $ echo Where are you going\?
    Where are you going?
2   $ echo Start on this line and \
    > go to the next line.
    Start on this line and go to the next line.
3   $ echo \\
    \
4   $ echo '\\'
    \\
5   $ echo '\$5.00'
    \$5.00
6   $ echo  "\$5.00"
    $5.00
7   $ echo 'Don\'t you need $5.00?'
    >
    >'
    Don\t you need .00?
```

说明

1. 反斜杠阻止 shell 对问号执行文件名替换。
2. 反斜杠转义换行符，让下一行能成为当前行的一部分。
3. 反斜杠本身也是特殊字符，因此它阻止 shell 解释跟在它后面的那个反斜杠。
4. 括在单引号里的反斜杠不会被解释。
5. 单引号里的所有字符都被当成字面字符。此处的单引号没有任何特殊目的。
6. 括在双引号里时，反斜杠保护美元符不因变量替换而被解释。
7. 单引号内的反斜杠不会被解释。因此，shell 将看到 3 个单引号(串尾那个未能匹配成对)。屏幕上出现次提示符，等待用户输入用于结束的单引号。如果最终输入了这个引号，shell 就将剔除该字符串中的全部引号，然后将它传给 echo 命令。因为前两个引号匹配成对了，所以字符串的剩余部分 t you need $5.00?没有被括在引号中。shell 将试着对$5 求值，结果为空，因此打印出.00。

13.11.2　单引号

单引号必须匹配成对。它们能保护所有元字符不被解释。要打印单引号，就必须用双引号把它括起来，或者用反斜杠转义它。

范例 13-73

```
1   $ echo 'hi there
    > how are you?
    > When will this end?
    > When the quote is matched
    > oh'
    hi there
    how are you?
    When will this end?
```



```
   When the quote is matched
   oh
2  $ echo Don\'t you need '$5.00?'
   Don't you need $5.00?
3  $ echo 'Mother yelled, "Time to eat!"'
   Mother yelled, "Time to eat!"
```

说明

1. 单引号没能在这一行内匹配，所以 bash shell 显示一个次提示符，等着引号被匹配。

2. 单引号保护所有的元字符不被解释。“Don't” 里的引号被反斜杠转义(反斜杠保护的对象是单个字符，而不是一个串)。否则它就会去匹配第一个单引号，导致字符串末尾的那个单引号配不成对。$和?被括在一对单引号中，不让 shell 解释它们。即，将它们当成字面字符。

3. 单引号保护字符串中的双引号不被 shell 解释。

13.11.3 双引号

双引号必须匹配成对。双引号允许对它所括的内容进行变量替换和命令替换，同时保护其他的特殊元字符不被 shell 解释。

范例 13-74

```
1  $ name=Jody
2  $ echo "Hi $name, I'm glad to meet you!"
   Hi Jody, I'm glad to meet you!
3  $ echo "Hey $name, the time is $(date)"
   Hey Jody, the time is Wed Jul 14 14:04:11 PST 2004
```

说明

1. 将变量 name 赋值为字符串 Jody。

2. 字符串两端的双引号将保护所有的特殊元字符不被 shell 解释，$name 中的$是个例外。这个例子在双引号里执行了变量替换。

3. 变量替换和命令替换都可以在双引号中执行。这个例子中，变量 name 被展开，反引号中的 date 命令也被执行(请参见下面的“命令替换”)。

13.12 命令替换

命令替换的用处是将命令的输出结果赋给一个变量，或将命令的输出结果代入字符串。所有的 shell 都使用反引号来执行命令替换[21]。bash 允许使用两种格式：在旧的格式中，命令被放在反引号内，而在新的 Korn 风格的格式中，命令被放在前面有个美元符的一对括号内。

---

[21] bash shell 将反引号用于命令替换是为了向上兼容，它还有另一种替换方法。

执行扩展时，bash 先执行命令，然后返回命令的标准输出，输出结果末尾的换行符都将被删除。使用旧风格的反引号格式进行替换时，反斜杠将保留它的字面含意，除非它后面跟的是$、`或\。当使用$(command)格式时，括号内的所有字符只用于构成命令，都不会被特殊对待。

命令替换可以被嵌套。当用旧格式嵌套时，内部的反引号必须用反斜杠来转义。

格式

```
'UNIX command'   # Old method with backquotes
$(UNIX command)  # New method
```

范例 13-75

(老的方式)

```
1   $ echo "The hour is `date +%H`"
    The hour is 09
2   $ name=`awk -F: '{print $1}' database`
    $ echo $name
    Ebenezer Scrooge
3   $ "  ls 'ls /etc'
    shutdown
4   $ set `date`
5   $ echo $*
    Wed Jul 14 09:35:21 PDT 2004
6   $ echo $2 $6
    Jul 2004
7   $ echo `basename \`pwd\``
    ellie
```

说明

1. date 命令的输出被替换到串中。

2. 将 awk 命令的输出赋给变量 name，并显示出来。

3. 反引号里那条 ls 命令的输出是/etc 目录下的文件列表。这些文件名将成为第一个 ls 命令的参数。当前目录下在/etc 里同名的文件都被列了出来(在当前目录下未找到同名文件将产生一个错误信息，如 ls:termcap:No such file or directory)。

4. set 命令把 date 命令的输出赋给位置参量。空白符拆分词表，并将各个词放到对应的参数中。

5. $*变量保存所有的参数。date 命令的输出被存放在$*中。用空白符分隔各个参数。

6. 打印第 2 个和第 6 个参数。

7. 为了把变量 dirname 设置成当前工作目录名(不含各级父目录)，使用了嵌套的命令替换。首先执行 pwd 命令，取得当前工作目录的全路径名，将其作为参数传递给 UNIX 命令 basename。basename 命令删除路径名中最后一个元素以外的所有元素。在反引号内嵌套命令时，需要用一个反斜杠来转义内层命令的反引号。

下面的范例 13-76 给出了 bash 中用来代替反引号进行命令替换的方法。



范例 13-76

(新方式)

```
1  $ d=$(date)
   $ echo $d
   Wed Jul 14 09:35:21 PDT 2004
2  $ lines = $(cat filex)
3  $ echo The time is $(date +%H)
   The time is 09
4  $ machine=$(uname -n)
   $ echo $machine
   jody
5  $ pwd
   /usr/local/bin
   $ dirname="$(basename $(pwd)) "   # Nesting commands
   $ echo $dirname
   bin
6  $ echo $(cal)                     # Newlines are lost
   July 2004 S M Tu W Th F S 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
   16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
7  $ echo "$(cal)"
       July 2004
   S  M  Tu  W   Th  F  S
                 1   2  3
   4  5   6  7   8   9  10
  11 12  13 14  15  16  17
  18 19  20 21  22  23  24
  25 26  27 28  29  30  31
```

说明

1. date 命令被括在括号中。命令的输出被替换进表达式，然后赋给变量 d，并显示出来。

2. 把 cat 命令的输出赋给变量 lines。

3. 再次将 date 命令括在括号中。date +%H 的输出(当前小时数)被替换到表达式中并回显到屏幕上。

4. 将 uname -n 的输出(主机名)赋给变量 machine。machine 变量的值被回显到屏幕上。

5. pwd 命令的输出(当前工作目录)是/usr/local/bin。给变量 dirname 赋值为命令替换的结果输出，这里命令替换是嵌套的。$(pwd)是执行的第一个命令替换。pwd 命令的输出被替换到表达式，然后 basename 程序将替换的结果，/usr/local/bin，作为它的参数，其结果为 basename /usr/local/bin。

6. cal(当前月份)命令的输出被回显。执行命令替换时最后的换行符被删除。

7. 整个命令替换表达式放到双引号中时，最后的换行符被保留，这样得到的日历看起来就像个样子了。

13.13 算术扩展

shell 通过对一个算术表达式求值并替换结果来执行算术扩展。表达式可以像在双引号中一样来进行处理，并且可以嵌套。关于算术操作和算术求值的详细讨论，请参见 12.3.4 “算术操作符和 let 命令”。

求值算术表达式有以下两种格式。

格式

```
$[ expression ]
$(( expression ))
```

范例 13-77

```
echo $[ 5 + 4 - 2 ]
7
echo $[ 5 + 3 * 2]
11
echo $[(5 + 3) * 2]
16
echo $(( 5 + 4 ))
9
echo $(( 5 / 0 ))
bash: 5/0: division by 0 ( error token is "0")
```

13.14 扩展顺序

当您正在执行变量、命令、算术表达式和路径名的扩展时，shell 被设计成按照指定的顺序来扫描命令行。假设变量没有被引用，处理就将按下面的顺序进行。

(1) 花括号扩展
(2) 代字符号扩展
(3) 参量扩展
(4) 变量替换
(5) 命令替换
(6) 算术扩展
(7) 词分离
(8) 路径名扩展

13.15 数组

在 2.x 版本的 bash 中可以创建一维数组。数组允许将一列词放到一个变量名中，例如



一列数、一列名称或一列文件。数组可以用内置函数 declare –a 来创建，或者直接给变量名一个下标来创建，如 x[0]=5。索引值是从 0 开始的整数。数组没有上限，索引也不必是有序的数，如，x[0]、x[1]、x[2]等。要取出数组中的一个元素，命令语法为${数组名[索引]}。如果 declare 命令带-a 和-r 选项，将创建一个只读数组。

格式

```
declare -a variable_name
variable = ( item1 item2 item3 ... )
```

范例 13-78

```
declare -a nums=(45 33 100 65)
declare -ar names   (array is readonly)
names=( Tom  Dick  Harry)
states=( ME  [3]=CA  CT )
x[0]=55
n[4]=100
```

说明

给一个数组赋值时，索引将自动从 0 开始，每增加一个元素，索引就自动加 1。给数组赋值时不一定要提供索引，提供的索引也不必是有序的。要删除整个数组，可以用 unset 命令，后跟数组名，要删除数组中一个元素，则可在 unset 后跟上“数组名[下标]”。

declare、local 和 read-only 内置命令也可以带-a 选项来声明一个数组。带-a 选项的 read 命令用来从标准输入读取一列词到数组元素中。

范例 13-79

```
1   $ declare -a friends
2   $ friends=(Sheryl Peter Louise)
3   $ echo ${friends[0]}
    Sheryl
4   $ echo ${friends[1]}
    Peter
5   $ echo ${friends[2]}
    Louise
6   $ echo "All the friends are ${friends[*]}"
    All the friends are Sheryl Peter Louise
7   $ echo "The number of elements in the array is ${#friends[*]}"
    The number of elements in the array is 3
8   $ unset friends  or unset ${friends[*]}
```

说明

1. declare 内置命令用来明确地声明一个数组，但它不是必须的。当给任何一个使用下标的变量(如变量[0])赋值时，都将自动被当作是一个数组。

2. 给数组 friends 赋值：Shery1、Peter 和 Louise。

3. 通过将数组名和它的下标括在花括号中，并以 0 作为下标的值来访问 friends 数组的第 1 个元素。打印出 Shery1。

4. 用索引值 1 访问 friends 数组的第 2 个元素。

5. 用索引值 2 访问 friends 数组的第 3 个元素。

6. 将星号放在下标中时，可以访问数组中的所有元素。该行显示 friends 数组中的所有元素。

7. 语法${#friends[*]}输出的是数组的大小，即，数组中元素的个数。另一方面，${#friends[0]}将输出数组中第 1 个元素的字符个数。Shery1 中有 6 个字符。

8. unset 内置命令删除整个数组。通过键入 unset friends[1]将仅删除数组中的一个元素。它将删除 Shery1。

范例 13-80

```
1   $ x[3]=100
    $ echo ${x[*]}
    100
2   $ echo ${x[0]}
3   $ echo ${x[3]}
    100
4   $ states=(ME  [3]=CA  [2]=CT)
    $ echo ${states[*]}
    ME CA CT
5   $ echo ${states[0]}
    ME
6   $ echo ${states[1]}
7   $ echo ${states[2]}
    CT
8   $ echo ${states[3]}
    CA
```

说明

1. 数组 x 的第 3 个元素被赋值为 100。索引数为 3 是可用的，但因为前两个元素还不存在，数组大小仅为 1。${x[*]}显示数组 x 的一个元素。

2. x[0]没有值，x[1]和 x[2]也没有。

3. x[3]的值是 100。

4. states 数组的索引 0 被赋值 ME，索引 3 被赋值 CA，索引 2 被赋值 CT。在这个例子中，可以看出 bash 并不介意把值保存在哪个索引里，并且索引数也没有必要连续。

5. 打印 states 数组的第 1 个元素。

6. states[1]中没保存任何内容。

7. states 数组的第 3 个元素，states[2]，被赋值 CT。

8. states 数组的第 4 个元素，states[3]，被赋值 CA。

13.16 函数

bash 函数用于在当前 shell 环境(不派生一个子进程)中通过函数名来执行一组命令。它们与脚本很相似，但是效率更高。函数一经定义，就成了 shell 内存映像的一部分，因此，



调用函数时，shell 不必像使用(脚本)文件那样读取磁盘。函数常常被用来提高脚本的模块化程度。函数定义之后，可以被重复调用。虽然当交互运行时可以在提示符下定义，但函数常常定义在用户的初始化文件.bash_profile 中。注意，函数必须在定义之后才能被调用。

13.16.1 定义函数

有两种格式可以声明一个 bash 函数。一种是旧的 Bourne shell 方式，函数名后面跟有一对空的圆括号，再跟函数定义。新的格式(Korn shell 方式)是用 function 关键字后跟函数名以及函数定义。如果使用新方法，圆括号是可选的。函数的定义由花括号中的一组命令构成，命令之间以分号分隔。最后那条命令必须以分号终结。花括号两侧的空格是必需的。传递给函数的任何参数被当作函数内的位置参量。一个函数的位置参量对函数来说是局部的。内置函数 local 允许在函数定义中创建局部变量。函数还可以递归，可以无限次调用它本身。

格式

```
function_name () { commands ; commands; }
function function_name { commands ; commands; }
function function_name () { commands ; commands; }
```

范例 13-81

```
1  $ function greet { echo "Hello $LOGNAME, today is $(date)"; }
2  $ greet
   Hello ellie, today is Wed Jul 14 14:56:31 PDT  2004
3  $ greet () { echo "Hello $LOGNAME, today is $(date)"; }
4  $ greet
   Hello ellie, today is Wed Jul 14 15:16:22 PDT  2004
5  $ declare -f
   declare -f greet()
   {
     echo "Hello $LOGNAME, today is $(date)"
   }
6  $ declare -F㉒
   declare -f greet
7  $ export -f greet
8  $ bash                 # Start subshell
9  $ greet
   Hello ellie, today is Wed Jul 14 17:59:24 PDT  2004
```

说明

1. 关键字 function 后跟函数名 greet。函数定义用花括号括起来。在左花括号后必须有一个空格。同一行的语句用分号来终结。

2. 当调用 greet 函数时，括在花括号中的命令在当前 shell 环境下执行。

3. 用 Bourne shell 格式再次定义 greet 函数，函数名后跟一对空的圆括号，然后是函数定义。

㉒ 只有 bash 2.x 版本支持。

4. greet 函数再次被调用。
5. 带-f 开关的 declare 命令列出在该 shell 中定义的所有函数以及它们的定义。
6. 带-F 开关的 declare 命令只列出函数名。
7. 带-f 开关的 export 命令使函数成为全局的。即子 shell 也可以获得。
8. 启动一个新的 bash shell。
9. 在这个子 shell 中函数是定义的，因为它已经被导出。

范例 13-82

```
1   $ function fun {
        echo "The current working directory is $PWD."
        echo "Here is a list of your files: "
        ls
        echo "Today is $(date +%A).";
    }
2   $ fun
    The current working directory is /home.
    Here is a list of your files:
    abc      abc123   file1.bak   none      nothing   tmp
    abc1     abc2     file2       nonsense  nowhere   touch
    abc122   file1    file2.bak   noone     one
    Today is Wednesday.
3   $ function welcome { echo "Hi $1 and $2"; }
4   $ welcome tom joe
    Hi tom and joe
5   $ set jane anna lizzy
6   $ echo $*
    jane anna lizzy
7   $ welcome johan joe
    hi johan and joe
8   $ echo  $1 $2
    johan joe
9  $ unset -f welcome     # unsets the function
```

说明

1. 命名并定义函数 fun。关键字 function 后跟一个函数名和括在花括号内的一组命令。命令分列在单独的行。如果它们列在同一行里，就必须用分号来分隔。在第一个花括号后必须有一个空格，否则将出现语法错误。必须在一个函数使用前就定义它。

2. 当被调用时，函数的行为和脚本的相同。依次执行函数定义中的每条命令。

3. 有两个位置参量用于函数 welcome。当参数传递给函数时，将给位置参量赋实际值。

4. 函数的参数 tom 和 joe 被分别赋给$1 和$2。一个函数里的位置参量是函数私有的，而且不会影响函数外的任何使用。

5. 在命令行设置位置参量。这些变量对在函数中设置的那些没有任何影响。

6. $*显示当前设置的位置参量值。

7. 调用函数 welcome。赋给位置参量的值是 johan 和 joe。



8. 在命令行赋值的位置参量不受在函数中定义的那些参数的影响。
9. 带-f 开关的 unset 内置命令将清除函数定义。

13.16.2 列出和清除函数

要列出函数和它们的定义，就要用 declare 命令。在 bash 2.x 及以上版本中，declare –F 仅列出函数名。函数及其定义将和输出变量、局部变量一起显示在输出结果中。函数及其定义可以用 unset -f 命令来清除。

13.17 标准 I/O 和重定向

shell 启动时继承了 3 个文件：stdin、stdout 和 stderr。标准输入通常来自键盘。标准输出和标准错误输出通常被发往屏幕。有些时候，需要从文件读取输入，或者将输出结果和报错信息写入文件。这些都可以通过 I/O 重定向来实现。请参见表 13-23 中列出的重定向操作符。

表 13-23 重定向

重定向操作符	功 能
< 文件名	重定向输入
> 文件名	重定向输出
>>文件名	追加输出
2>文件名	重定向标准错误输出
2>>文件名	重定向和追加标准错误输出
&>文件名	重定向标准输出和标准错误输出
>&文件名	重定向标准输出和标准错误输出(首选方式)
2>&1	将标准错误输出重定向到输出的去处
1>&2	将输出重定向到标准错误输出的去处
>|	重定向输出时忽略 noclobber
<>文件名	如果是一个设备文件(/dev)，使用文件作为标准输入和标准输出

范例 13-83

```
1   $ tr '[A-Z]'  '[a-z]' < myfile     # Redirect  input
2   $ ls > lsfile                      # Redirect output
    $ cat lsfile
    dir1
    dir2
    file1
    file2
    file3
```

```
3  $ date >> lsfile                # Redirect and append otuput
   $ cat lsfile
   dir1
   dir2
   file1
   file2
   file3
   Sun Sept 17 12:57:22 PDT 2004
4  $ cc prog.c 2> errfile          # Redirect error
5  $ find . -name \*.c -print > foundit 2> /dev/null
   # Redirect output to foundit and errors to /dev/null,
   # respectively.
6  $ find . -name \*.c -print >& foundit
   # Redirect both output and errors to foundit.
7  $ find . -name \*.c -print > foundit 2>&1
   # Redirect output to foundit and send errors to where output
   # is going; i.e. foundit
8  $ echo "File needs an argument" 1>&2
   # Send standard output to error
```

说明

1. UNIX/Linux 命令 tr 的标准输入被重定向为来自文件 myfile，而不从键盘获取输入。所有大写字母被转换为小写字母。

2. ls 命令将它的输出重定向到文件 lsfile，不再把输出结果发往屏幕。

3. date 命令的输出结果因重定向被追加到文件 lsfile 中。

4. 编译 C 程序源文件 prog.c。如果编译失败，标准错误输出被重定向到文件 errfile。您就可以拿这个错误信息文件去请教身边的高手，请他给出相应的解释。

5. find 命令开始在当前工作目录下查找以.c 结尾的文件名，将找到的文件名打印到文件 foundit 中。find 命令输出的错误信息则被发往/dev/null。

6. find 命令开始在当前工作目录下查找以.c 结尾的文件名，将找到的文件名打印到文件 foundit 中。find 命令输出的错误信息也写进 foundit。

7. 同 6 中的解释。

8. echo 命令将它的信息发往标准错误输出。该命令的标准输出被合并到标准错误输出中。

exec 命令和重定向

exec 命令能够在不启动新进程的前提下，将当前正在运行的程序替换为一个新的程序。使用 exec 命令，不需创建子 shell 就能改变标准输入和输出(参见表 13-24)。用 exec 打开文件后，read 命令每次都会将文件指针移到文件的下一行，直到文件末尾。如果要再次从头开始读文件，就必须先关闭文件再打开。但是，如果使用 cat 和 sort 这类 UNIX 工具，操作系统会在命令结束后自动关闭文件。

表 13-24 exec 命令

Exec 命令	功能
exec ls	ls 将顶替 shell 运行。ls 运行结束后，它启动时所在的 shell 不会返回运行状态
exec <filea	打开文件 filea，用于读标准输入
exec >filex	打开文件 filex，用于写标准输出
exec 3<datfile	打开文件 datfile，将其作为文件描述符 3，用于读取输入
sort <&3	将文件 datfile 排序
exec 4>newfile	打开文件 newfile，将其作为文件描述符 4，用于写输出
ls >&4	将 ls 的输出结果重定向到 newfile
exec 5<&4	将文件描述符 5 变成文件描述符 4 的一个副本
exec 3<&-	关闭文件描述符 3

范例 13-84

```
1    $ exec date
     Thu Oct 14 10:07:34  PDT 2004
      <Login prompt appears if you are in your login shell >
2    $ exec > temp
     $ ls
     $ pwd
     $ echo Hello
3    $ exec > /dev/tty
4    $ echo Hello
    Hello
```

说明

1. exec 命令在当前 shell 中执行 date 命令(不另外派生一个子 shell)。由于 date 命令是顶替当前 shell 执行，所以，当 date 命令退出后，shell 便终止了。如果这个 bash shell 是从另一个 TC shell 中启动的，结果将是 bash shell 退出，并在屏幕上出现 TC shell 的提示符。如果是在登录 shell 中尝试这个例子，结果将是退出系统。如果在 shell 窗口中交互式运行这个例子，结果则是窗口被关闭。

2. exec 命令打开文件 temp 作为当前 shell 的标准输出。此后，ls、pwd 和 echo 的输出结果将不再发往屏幕，而是写入文件 temp(参见图 13-2)。

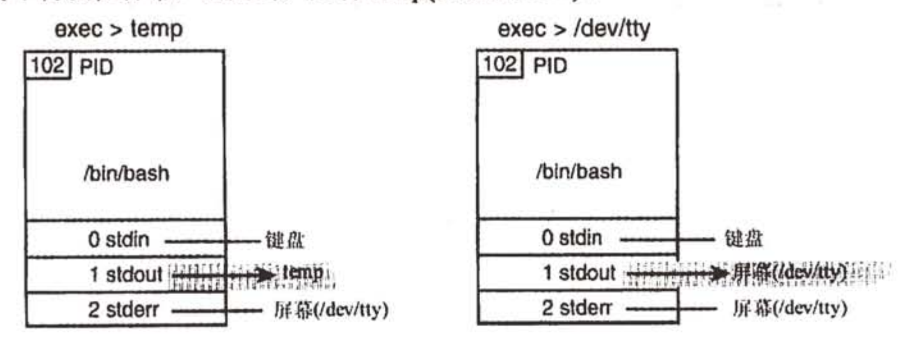


图 13-2 exec 命令和文件描述符

3. exec 命令将标准输出重新指向终端。现在，输出结果将发到屏幕，就如第 4 行显示的那样。

4. 标准输出再次指向终端(/dev/tty)。

范例 13-85

```
1   > bash
2   $ cat doit
    pwd
    echo hello
    date
3   $ exec < doit
    /home/homebound/ellie/shell
    hello
    Thu Oct 14 10:07:34  PDT 2004
4   >
```

说明

1. bash 从一个 TC shell 提示符下启动(这样做的目的是因为当 exec 命令退出时，用户不会退出系统)。

2. 显示文件 doit 的内容。

3. 命令 exec 打开文件 doit 作为标准输入。shell 将从该文件而不是键盘读取输入。exec 执行文件 doit 中的命令，替换掉当前 shell。当最后一条命令退出时，shell 也随之退出。

4. exec 命令完成后，bash shell 退出，屏幕上出现了 TC shell 的提示符，这个 TC shell 是刚才那个 bash shell 的父 shell。假如您之前是在登录 shell 中，那么，当 exec 结束时，你将被注销。如果是在窗口中，结果将会是窗口关闭。

范例 13-86

```
1   $ exec 3> filex
2   $ who >& 3
3   $ date >& 3
4   $ exec 3>&-
5   $ exec 3<filex
6   $ cat <&3
    ellie    tty1   Jul 21 09:50
    ellie    ttyp1  Jul 21 11:16  (:0.0)
    ellie    ttyp0  Jul 21 16:49  (:0.0)
    Wed Jul 21 17:15:18 PDT 2004
7   $ exec 3<&-
8   $ date >& 3
    date: write error: Bad file descriptor
```

说明

1. 将文件描述符 3(fd 3)指定给文件 filex，然后打开它，并将输出重定向到该文件。参见图 13-3(a)。



exec 3>filex

101 /bin/bash	
0	stdin
1	stdout
2	stderr
3	filex

a. 用来写

exec 3<filex

102 /bin/bash	
0	stdin
1	stdout
2	stderr
3	filex

b. 用来读

图 13-3　exec 和文件描述符

2. 将 who 命令的输出结果发往 fd 3，即文件 filex。

3. 将 date 命令的输出结果发往 fd 3。由于文件 filex 已被打开，所以结果被追加到 filex 的末尾。

4. 关闭 fd 3。

5. exec 命令打开 fd 3 用于读输入。输入将被重定向到 filex。参见图 13-3(b)。

6. cat 程序从 fd 3 读取输入，fd 3 已被指定给文件 filex。

7. exec 命令关闭 fd 3(实际上，一读到文件末尾，操作系统就会关闭这个文件)。

8. 试图将 date 命令的输出写到 fd 3，bash 报告一个错误条件，因为文件描述符 3 已经被关闭了。

13.18　管道

管道(pipe)将管道符左侧这条命令的输出发送到管道符右侧那条命令的输入。一条管道线(pipeline)可能包括不止一个管道。

范例 13-87 中这些命令的目的是计算当前登录用户的数目，将 who 命令的输出结果保存到一个文件(tmp)，用 wc –l 计算文件 tmp 中的行数(wc –l)，然后删除文件 tmp(即算出当前的登录用户数)。

范例 13-87

```
1    $ who > tmp
2    $ wc -l tmp
     4 tmp
3    $ rm tmp
# Using a pipe saves disk space and time.
4    $ who | wc -l
     4
5    $ du .. | sort -n | sed -n '$p'
     1980   ..
6    $ ( du / | sort -n | sed -n '$p' ) 2> /dev/null
     1057747  /
```

说明

1. 将 who 命令的输出重定向到文件 tmp。

2. 命令 wc –l 显示文件 tmp 的行数。

3. 删除文件 tmp。

4. 使用管道功能，您就能用一个步骤完成前面这 3 个步骤的全部工作。who 命令的输出被发送到内核中某个不知名的缓冲区，wc –l 命令读这个缓冲区，然后将它的输出发到屏幕上。见图 13-4。

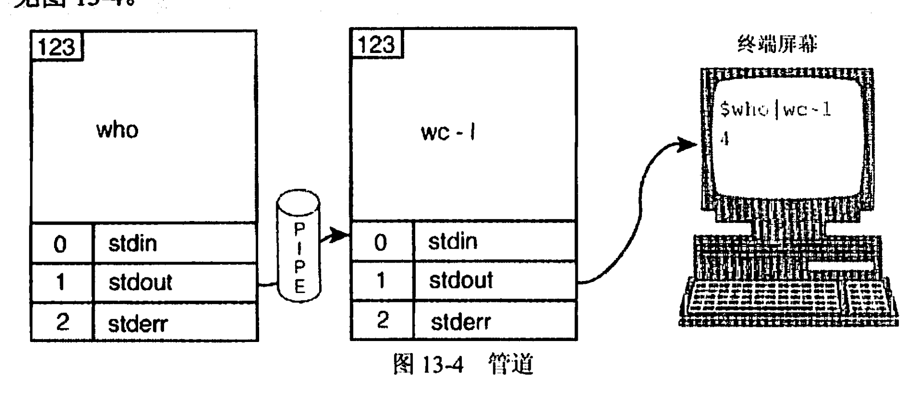


图 13-4　管道

5. du 命令的输出(即每个目录占用磁盘块的数目)，在父目录下开始(..)，经管道发给 sort 命令，按数值大小对其排序。之后，排序的结果又经管道发到 sed 命令，sed 命令打印所收到的输出的最后一行。见图 13-5。

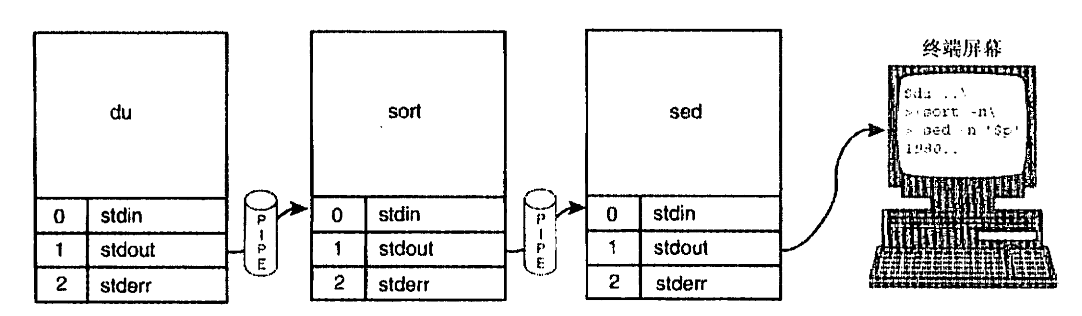


图 13-5　多个管道(过滤器)

6. 如果因为许可被关闭而无法进入一个目录，du 命令(在根目录下开始)将发送错误消息到标准错误输出(屏幕)。当您将整个命令行放到一对圆括号内时，所有的输出将发送到屏幕，且所有的错误被定向到 UNIX/Linux 的位存储桶/dev/null。

here 文档和重定向输入

here 文档是一种特殊格式的引用。here 文档为需要输入数据的程序(如 mail、sort 或 cat)接收内置文本，直至收到用户自定义的终止符。here 文档常常被 shell 脚本用来生成菜单。接收输入的命令后面跟着一个<<符号，<<符号后面是一个用户定义的词或符号，然后是换行符。接下来的文本行就是将要发送给程序的输入行。当用户定义的那个词或符号出现在某一行最左端(前后都不能有空格)，并且独占该行时，输入就终止了。这个词替代了用



Ctrl+D 组合键来通知程序停止读取输入。

如果定义终止符时，它前面的运算符是<<-，则输入的最后一行上，终止符前面可以出现制表符，但只能是制表符。匹配用户定义的终止词或终止符号必须做到一模一样。下面这个例子通过演示在命令行使用 here 文档来说明它的语法。其实，脚本中的 here 文档要实用得多。

范例 13-88

```
1    $ cat << FINISH                     # FINISH is a user-defined terminator
2    > Hello there $LOGNAME
3    > The time is $(date +%T).
     > I can't wait to see you!!!
4    > FINISH                            # terminator matches first
5    Hello there ellie                   # FINISH on line 1.
     The time is 19:42:12.
     I can't wait to see you!!!
6    $
```

说明

1. UNIX 的 cat 程序将一直接收输入，直到出现独占一行的 FINISH 为止。

2. 出现次提示符。接下来的文本是给 cat 命令的输入。shell 执行 here 文档中的变量替换。

3. 执行 here 文档中的命令替换$(date +%T)。也可以用命令替换的旧格式：`date +T`。

4. 用户自定义的终止符 FINISH 标志着 cat 程序的输入到此结束。终止符必须独占一行，而且前后都不能出现空格。

5. 显示 cat 程序的输出。

6. shell 提示符重新出现。

范例 13-89

```
1    $ cat <<- DONE
     >   Hello there
     >       What's up?
     >Bye now The time is `date`.
2    >     DONE
       Hello there
           What's up?
   Bye now The time is Sun Feb 819:48:23 PST 2004.
   $
```

说明

1. cat 程序接收输入，直到出现自成一行的 DONE 为止。操作符<<-允许输入和输入的终止符前面有一或多个制表符。如果在命令行键入这个例子可能会因为 Tab 键而产生错误。如果从一个脚本运行该例，它将不会有问题。

2. 最后匹配到的终止符 DONE 前面有一个制表符。cat 程序的输出显示在屏幕上。

13.19 shell 调用选项

当 shell 用 bash 命令启动时，它可以带上选项来更改它的行为。有两种类型的选项：单字符选项和多字符选项。单字符选项由一个破折号和一个字符组成。多字符选项由两个破折号和任意数目的字符组成。多字符选项必须出现在单字符选项之前。一个交互式的登录 shell 启动时通常添加了-i(启动一个交互式的 shell)，-s(从标准输入读取)和-m(打开作业控制)选项。参见表 13-25。

表 13-25 bash 2.x shell 调用选项

选 项	含 义
-c string	从 string 中读取命令。string 后的任何参数都赋给位置参量，从$0 开始
-D	将前面有一个$的由双引号括着的字符串打印到标准输出。当当前 locale 不是 C 或 POSIX 时，这些串服从于语言转换
-i	Shell 是在交互模式下。忽略 TERM、QUIT 和 INTERRUPT
-s	从标准输入读取命令并且允许设置位置参量
-r	启动一个受限制的 shell
--	用信号通知选项的结束并禁止更多的选项处理。--或-后的任何参数被当作文件名和参数
--dump-strings	同-D
--help	显示一个内置命令的使用信息并退出
--login	将 bash 作为一个登录 shell 调用
--noediting	当 bash 交互式运行时，不使用 readline 库
--noprofile	bash 启动时不读取初始化文件：/etc/profile、~/.bash_profile、~/.bash_login 或~/.profile
--norc	对于交互式 shell，bash 不读取~/.bashrc 文件。如果运行的 shell 是 sh，则默认是打开的
--posix	改变 bash 的行为以符合 POSIX 1003.2 标准，否则不改变
--quiet	默认设置，在 shell 启动时不显示任何消息
--rcfile file	如果 bash 是交互式的，使用该初始化文件代替~/.bashrc
--restricted	启动一个受限制的 shell
--verbose	打开 verbose。和-v 一样
--version	显示该 bash shell 的版本信息并退出

13.19.1 set 命令和选项

set 命令可用来打开或关闭 shell 选项，就像处理命令行参数一样。要打开一个选项，在选项前加一个破折号(-)。要关闭一个选项，在选项前加一个加号(+)。表 13-26 列出了 set

选项的清单。

范例 13-90

```
1  $ set -f
2  $ echo *
   *
3  $ echo ??
   ??
4  $ set +f
```

说明

1. 打开 f 选项，禁止文件名扩展。
2. 星号没有被扩展。
3. 问号没有被扩展。
4. f 选项被关闭。文件名扩展打开。

表 13-26 内置 set 命令选项

选项名	快捷开关	含义
allexport	-a	从这个选项被设置开始就自动标明要输出的新变量或修改过的变量，直至选项关闭
braceexpand	-B	打开花括号扩展，是一个默认设置
emacs		使用 emacs 内置编辑器进行命令行编辑，是一个默认设置
errexit	-e	当一个命令返回一个非零退出状态时，退出。读取初始化文件时不设置
histexpand	-H	当执行历史替换时打开!和!!，是一个默认设置
history		打开命令行历史。默认时为开
ignoreeof		禁止用 EOF(Ctrl+D)键退出 shell。必须键入 exit 才能退出。和设置 shell 变量 IGNOREEOF=10 一样
keyword	-k	为命令把 keyword 变量加到环境中
interactive-comments		对于交互式 shell，#用来将后面的文本作为注释。
monitor	-m	允许作业控制
noclobber	-C	防止文件在重定向时被重写
noexec	-n	读命令，但不执行。用来检查脚本的语法。交互式运行时不设置
noglob	-d	禁止用路径名扩展。即，关闭通配符
notify	-b	通知用户什么时候后台作业完成
nounset	-u	如果一个变量在扩展时没被设置就显示一个错误
onecmd	-t	在读取和执行命令后退出
physical	-P	设置时，在键入 cd 或 pwd 时禁止符号链接。用物理目录替代
posix		如果默认操作不符合 POSIX 标准就改变 shell 行为

(续表)

选 项 名	快捷开关	含 义
privileged	-p	设置时，shell 不读取.profile 或 ENV 文件，且 shell 函数不从环境继承。而自动为 setuid 脚本设置
verbose	-v	为调试打开 verbose 模式
vi		使用 vi 内置编辑器进行命令行编辑
xtrace	-x	为调试打开 echo 模式

13.19.2 shopt 命令和选项

shopt(bash 2.x)命令也可以用来打开或关闭 shell 选项。

表 13-27 shopt 命令选项

选 项	含 义
cdable_vars	如果给 cd 内置命令的参数不是一个目录，就假设它是一个变量名，变量的值是将要转换到的目录
cdspell	纠正 cd 命令中目录名的较小拼写错误。检查的错误包括颠倒顺序的字符，遗漏的字符以及重复的字符。如果找到一处修改，正确的路径将打印出，命令将继续。只用于交互式 shell
checkhash	bash 在试图执行一个命令前，先在哈希表中寻找，以确定命令是否存在。如果命令不存在，就执行正常的路径搜索
checkwinsize	bash 在每个命令后检查窗口大小，如果有必要，就更新 LINES 和 COLUMNS 的值
cmdhist	bash 试图将一个多行命令的所有行保存在同一个历史项中。这使得多行命令的重新编辑更方便
dotglob	bash 在文件名扩展的结果中包括以点(.)开头的文件名
execfail	如果一个非交互式 shell 不能执行指定给 exec 内置命令作为参数的文件，它不会退出。如果 exec 失败，一个交互式 shell 不会退出
expand_aliases	别名被扩展。默认为打开
extglob	打开扩展的模式匹配特性(正常的表达式元字符来自 Korn shell 的文件名扩展)
histappend	当 shell 退出时，历史清单将添加到以 HISTFILE 变量的值命名的文件中，而不是覆盖文件
histreedit	如果 readline 正被使用，用户有机会重新编辑一个失败的历史替换
histverify	如果设置，且 readline 正被使用，历史替换的结果不会立即传递给 shell 解释器。而是将结果行装入 readline 编辑缓冲区中，允许进一步修改
hostcomplete	如果设置，且 readline 正被使用，当正在完成一个包含@的词时 bash 将试图执行主机名补全。默认为打开
huponexit	如果设置，当一个交互式登录 shell 退出时，bash 将发送一个 SIGHUP(挂起信号)给所有的作业



(续表)

选　项	含　义
interactive_comments	在一个交互式 shell 中，允许以#开头的词以及同一行中其他的字符被忽略。默认为打开
lithist	如果打开，且 cmdhist 选项也打开，多行命令将用嵌入的换行符保存到历史中，而无需在可能的地方用分号来分隔
mailwarn	如果设置，且 bash 用来检查邮件的文件自从上次检查后已经被访问，将显示消息“The mail in mailfile has been read”
nocaseglob	如果设置，当执行文件名扩展时，bash 在不区分大小写的方式下匹配文件名
nullglob	如果设置，bash 允许没有匹配任何文件的文件名模式扩展成一个空串，而不是它们本身
promptvars	如果设置，提示串在被扩展后再经历变量和参量扩展。默认为打开
restricted_shell	如果 shell 在受限模式下启动就设置这个选项。该值不能被改变。当执行启动文件时不能复位该选项，允许启动文件发现 shell 是否是受限的
shift_verbose	如果该选项设置，当移动计数超出位置参量个数时，shift 内置命令将打印一个错误消息
sourcepath	如果设置，source 内置命令使用 PATH 的值来寻找包含作为参数提供的文件的目录。默认为打开
source	点(.)的同义词

13.20 shell 内置命令

shell 有许多内置到它的源码中的命令。因为命令是内置的，shell 无需到磁盘上定位它们，这样执行速度将快得多。bash 提供的 help 特性提供了所有内置命令的在线帮助。内置命令在表 13-28 中列出。

表 13-28　内置命令

命　令	含　义
:	空命令。返回退出状态零
.	在当前进程的环境下执行程序。同 source
.file	点命令读取并执行 file 里的命令
break	跳出最内层的循环
break [n]	参见 14.6 节，“循环控制命令”
alias	为存在的命令列出并创建别名
bg	将一个作业放到后台
bind	显示当前键和函数的绑定，或将键和一个 readline 函数或宏绑定
builtin [sh-builtin [args]]	运行一个 shell 内置命令，给它传递参数并返回退出状态 0。当一个函数和内置命令同名时很有用

(续表)

命　令	含　义
cd [arg]	如果没有参数，就将目录改变到主目录或改变到参数的值
command command [arg]	运行一个命令，当有一个函数和它同名时，忽略函数
continue [n]	参见 14.6 节，“循环控制命令”
declare [var]	显示所有的变量或用可选属性声明变量
dirs	显示 pushd 产生的当前记录的目录
disown	从作业表中删除一个活动的作业
echo [args]	显示用换行符终止的参数
enable	开启和关闭 shell 内置命令
eval [args]	读参数作为 shell 的输入，并执行产生的命令
exec command	执行命令来取代当前的 shell
exit [n]	以状态 n 退出 shell
export [var]	使 var 能被子 shell 识别
fc	用于编辑历史命令的历史编辑命令
fg	将后台作业放到前台
getopts	解析并处理命令行选项
hash	控制内部哈希表以更快地搜索命令
help [command]	显示关于内置命令的帮助信息，如果指定命令，将显示该内置命令的详细帮助
history	显示带行号的历史清单
jobs	列出放在后台的作业
kill [-singal process]	发送信号给指定 PID 号或作业号的进程。可在提示符下键入：kill –l
getopts	用于 shell 脚本以解析命令行并检查合法的选项
let	用来对算术表达式求值并将算术计算的结果赋给变量
local	用在函数中以限制变量在函数中的作用域
logout	退出登录 shell
popd	从目录栈中删除项
pushd	往目录栈中添加项
pwd	显示当前工作目录
read [var]	从标准输入读取一行到变量 var
readonly [var]	使变量 var 只读。不能被复位
return [n]	从一个函数返回，n 是返回的退出值
set	设置选项和位置参量。参见 14.4 节，“set 命令和位置参量”
shift [n]	向左移动位置参量 n 次
stop pid	终止 PID 号进程的执行
suspend	暂停当前 shell 的执行(如果是一个登录 shell 就不暂停)



(续表)

命　　令	含　　义
test	检查文件类型且测试条件表达式
times	为从该 shell 运行的进程显示所累积的用户和系统时间
trap [arg] [n]	当 shell 接收到信号 n(0、1、2 或 15)时执行参数
type [command]	打印命令的类型。例如，pwd 是一个内置 shell 命令
typeset	和 declare 一样。设置变量并给它们属性
ulimit	显示并设置进程资源限度
umask [octal digits]	设置创建文件时关于文件属主、属组和其他用户执行权限的掩码
unalias	删除别名
unset [name]	删除变量值或函数
wait [pid#n]	等待后台 PID 号为 n 的进程返回并报告终止状态

习题 48　Bash Shell 入门

1. 哪个进程把登录提示符显示到屏幕上?
2. 哪个进程为 HOME、LOGNAME 和 PATH 赋值?
3. 怎么才能知道自己正在运行哪种 shell?
4. 如何改变登录 shell?
5. 在哪里(哪个文件)指定您的登录 shell?
6. 解释/etc/profile 和~/.bash_profile 这两个文件之间的区别。shell 先执行哪一个?
7. 编辑.bash_profile 文件，完成下列功能:
   a) 欢迎用户。
   b) 如果路径中不包括主目录，将其加入。
   c) 用 stty 命令设置退格键的擦除功能。
   d) 键入：source .profile。source 命令的功能是什么?
8. BASH_ENV 文件是什么? 什么时候执行?
9. 默认的主提示符是什么?
   a) 改变提示符以包括当天的时间和主目录。
   b) 默认的次提示符是什么? 它的功能是什么?
10. 解释下面每项设置的功能:
    a) set –o ignoreeof
    b) set –o noclobber
    c) set –o emacs
    d) set –o vi
11. 前一个例子中的设置保存在哪个文件中? 它们为什么被保存在那里?
12. shopt　- p 做什么用? 为什么用 shopt 而不是 set 命令?
13. 什么是内置命令? 如何知道一个命令是内置命令还是可执行程序? 命令 builtin 的

作用是什么？命令 enable 呢？

14. 什么情况可以使 shell 返回退出状态 127？

习题 49 作业控制

1. 程序和进程有什么不同？什么是作业？
2. shell 的 PID 是什么？
3. 如何停止一个作业？
4. 什么命令可以将一个后台作业放到前台？
5. 如何列出所有正在运行的作业？如何列出所有暂停的作业？
6. kill 命令的作用是什么？
7. jobs -l 显示什么？kill -l 显示什么？

习题 50 命令补全、历史和别名

1. 什么是文件名补全？
2. 用于保存在命令行键入的命令历史的文件是什么？
3. HISTSIZE 变量控制什么？HISTFILESIZE 控制什么？
4. "bang，bang"是什么意思？
5. 如何重新执行最后一条以 v 开头的命令？
6. 如何重新执行第 125 条命令？如何反序打印历史清单？
7. 如何将所使用的交互式编辑器设置为 vi 编辑器？可以将该设置放在哪个初始化文件中？
8. fc 命令是什么？
9. readline 库的目的是什么？它从哪个初始化文件中读取指令？
10. 什么是按键绑定？如何确定绑定了哪些键？
11. 什么是全局参数？
12. 为下列命令创建别名：
    a) clear
    b) fc –s
    c) ls --color=tty
    d) kill –l

习题 51 shell 元字符

1. 创建一个名为 wildcards 的目录。cd 到该目录然后在命令行键入：

touch ab abc a1 a2 a3 all a12 ba ba.1 ba.2 filex filey AbC ABC ABc2 abc

2. 写出能实现下列功能的命令，并测试所写的命令。
    a) 列出所有名称以 a 开头的文件。
    b) 列出所有名称以至少一个数字结尾的文件。
    c) 列出所有名称以 a 或 A 开头的文件。
    d) 列出所有名称以句号跟一个数字结尾的文件。
    e) 列出所有名称中只包含两个字母的文件。



f) 列出所有名称为 3 个字母组成，且 3 个字母都大写的文件。

g) 列出所有名称以 10、11 或 12 结尾的文件。

h) 列出所有名称以 x 或 y 结尾的文件。

i) 列出所有名称以数字、大写字母或小写字母结尾的文件。

j) 列出所有名称不是以 a、b 或 B 开头的文件。

k) 删除名称为两个字符，并以 a 或 A 开头的文件。

习题 52　重定向

1. 与终端关联的 3 个文件流的名字是什么？

2. 什么是文件描述符？

3. 用什么命令来完成下面这些任务：

   a) 把 ls 命令的输出重定向到文件 lsfile。

   b) 重定向 date 命令的输出，将其追加到文件 lsfile 尾部。

   c) 把 who 命令的输出重定向到文件 lsfile，会出现什么结果？

4. 只输入 cp，会出现什么结果？

5. 如何把上面这个例子产生的报错信息保存到一个文件中？

6. 用 find 命令从父目录开始，找出所有类型为目录的文件。把标准输出保存到文件 found 中，把所有报错信息保存到文件 found.errs 中。

7. 取 3 条命令的输出，将它们重定向到文件 gottem_all 中？

8. 使用管道来运行 ps 和 wc 命令，查出当前正在运行的进程数。

习题 53　变量

1. 什么是位置参量？在命令行键入：

set dogs cats birds fish

   a) 如何列出所有的位置参量？

   b) 哪个位置参量赋值为 birds？

   c) 如何输出位置参量的个数？

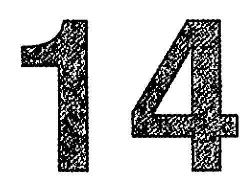
   d) 怎么从 shell 的内存中删除所有的位置参量？

2. 环境变量是什么？用来列出它们的命令是什么？创建一个名为 CITY 的环境变量并将它赋值为一个城市名。如何导出这个变量？

3. 局部变量是什么？将一个局部变量设成您的名字。打印它的值。然后删除它。

4. declare -i 的功能是什么？

5. $$变量显示什么？$!呢？





bash shell 编程

14.1 简介

当命令不在命令行上执行，而是通过一个文件执行时，该文件就称为 shell 脚本，脚本以非交互的方式运行。当 bash shell 以非交互方式运行时，它先查找环境变量 BASH_ENV(或 ENV)，该变量指定了一个环境文件(通常是.bashrc)，然后从该文件开始执行。当读取了 BASH_ENV 文件后，shell 就开始执行脚本中的命令①。

创建 shell 脚本的步骤

shell 脚本通常在编辑器中编写，由命令及其注释组成，注释是跟在井号(#)后面的内容，用来对脚本进行注解。

第一行 位于脚本左上角的第一行会指出由哪个程序来执行脚本中的行。这一行通常称为 shbang 行，写作：

```
#!/bin/bash
```

“#!”又称为幻数，内核根据它来确定该用哪个程序来翻译脚本中的行。这一行必须是脚本顶端第一行。bash shell 还能接受一些参数，来控制 shell 的行为，14.11 一节中的表 14-8 列出了所有的 bash 选项。

注释 注释是跟在#号后的行。注释可以自成一行，也可以跟在脚本命令后面与命令共处一行。注释被用来对脚本作注解。有时候，如果没有注释，就很难理解脚本究竟可以用来做什么。尽管注释很重要，但是脚本中却经常只有很少的注释，甚至根本就没有注释。我们要尽量养成为所做的工作书写注释的习惯，不光方便别人，也方便自己，因为也许两天后您就记不起脚本的用途了。

可执行语句与 bash shell 结构 一个 bash shell 程序由一组 UNIX/Linux 命令、bash shell

① bash 以非交互方式运行时，如果带有选项-norc 或--norc，则不读取 BASH_ENV 或 ENV 文件。



命令、程序结构控制语句和注释组成。

使脚本可执行　在创建文件时，文件并没有被自动授予可执行的权限。如果要运行脚本，就必须赋予它可执行的权限。可以用 chmod 命令来打开脚本的执行权限。

范例 14-1

```
1   $ chmod +x myscript
2   $ ls -lF  myscript
    -rwxr-xr-x   1  ellie        0 Jul  13:00 myscript*
```

说明

1. mod 命令为用户、组及其他用户开启执行权限。

2. ls 命令的输出信息显示，所有用户都拥有 myscript 文件的执行权限。文件名末尾的星号表示它是一个可执行程序。

脚本会话　在下例中，用户将在编辑器中创建一个脚本。保存文件后，用户打开脚本的执行权限，然后执行它。如果程序中有任何错误，bash shell 将立刻作出反应。

范例 14-2

(脚本)

```
1   #!/bin/bash
2   # This is the first Bash shell program of the day.
    # Scriptname: greetings
    # Written by:  Barbara Bashful
3   echo "Hello $LOGNAME, it's nice talking to you."
4   echo "Your present working directory is `pwd`."
    echo "You are working on a machine called `uname -n`."
    echo "Here is a list of your files."
5   ls      # List files in the present working directory
6   echo  "Bye for now $LOGNAME. The time is `date +%T`!"
```

(命令行)

```
    $ greetings     # Don't forget to turn turn on x permission!
    bash: ./greetings: Permission denied.
    $ chmod +x greetings
    $ greetings  or  ./greetings
3   Hello barbara, it's nice talking to you.
4   Your present working directory is /home/lion/barbara/prog
    You are working on a machine called lion.
    Here is a list of your files.
5   Afile        cplus    letter     prac
    Answerbook   cprog    library    prac1
    bourne       joke     notes      perl5
6   Bye for now barbara. The time is 18:05:07!
```

说明

1. 脚本的第一行 #!/bin/bash 告诉内核将由哪个 shell 解释器来执行程序中的各行。本例中是由 bash 解释器执行。

2. 注释以#号开头，不被执行。注释可以独立成行，也可以紧跟在命令的后面。

3. shell 完成变量替换后，echo 命令在屏幕上显示字符串。

4. shell 完成命令替换后，echo 命令将在屏幕上显示字符串。

5. 执行 ls 命令。#号后的所有文本都是注释，将被 shell 忽略。

6. echo 命令显示双引号中的字符串。放在双引号中的变量和命令将被展开和替换，因此，这里引号不是必须的。

14.2 读取用户输入

14.2.1 变量

上一章我们谈到如何定义或取消变量，变量可被设置为当前 shell 的局部变量，或是环境变量。如果您的 shell 脚本不需要调用其他脚本，其中的变量通常设置为脚本内的局部变量(参见第 13.10 节“变量”)。

要获取变量的值，在美元符后跟变量名即可。shell 会对双引号内的美元符后的变量执行变量扩展，单引号中的美元符则不会被执行变量扩展。

范例 14-3

```
1   name="John Doe" or declare name="John Doe"   # local variable
2   export NAME="John Doe"    # global variable
3   echo "$name" "$NAME"      # extract the value
```

14.2.2 read 命令

read 命令是一个内置命令，用于从终端或文件读取输入(参见表 14-1)。read 命令读取一个输入行，直至遇到换行符。行尾的换行符在读入时将被转换成一个空字符。如果 read 命令后未跟变量名，读入的行将被赋给内置变量 REPLY。也可以用 read 命令来中断程序的运行，直至用户输入一个回车键。要知道如何有效地使用 read 命令从文件读取输入行，请参见 14.6 节的“循环控制命令”。如果带-r 选项，read 命令将忽略反斜杠/换行符对，而把反斜杠作为行的一部分。read 命令有 4 个控制选项：-a，-e，-p，-r[②]。

表 14-1 read 命令

格 式	含 义
read answer	从标准输入读取一行并赋值给变量 answer
read first last	从标准输入读取一行，直至遇到第一个空白符或换行符。把用户键入的第一个词存到变量 first 中，把该行的剩余部分保存到变量 last 中
read	标准输入读取一行并赋值给内置变量 REPLY
read –a arrayname	读入一组词，依次赋值给数组 arrayname[③]

② 选项-a，-e 和-p 只可用于 bash 2.x 版中。

③ 2.0 版以前的 bash 未实现该功能。



(续表)

格　式	含　义
read -e	在交互式 shell 命令行中启用编辑器。例如，如果编辑器是 vi，则可以在输入行时使用 vi 命令[③]
read –p prompt	打印提示符，等待输入，并将输入赋值给 REPLY 变量[③]
read –r line	允许输入包含反斜杠[③]

范例 14-4

(脚本)

```
    #!/bin/bash
    # Scriptname: nosy

    echo -e "Are you happy? \c"
1   read answer
    echo "$answer is the right response."
    echo -e "What is your full name? \c"
2   read first middle last
    echo "Hello  $first"
    echo -n "Where do you work? "
3   read
4   echo I guess $REPLY keeps you busy!
--------------------------------------------------------④
5   read -p "Enter your job title: "
6   echo "I thought you might be an $REPLY."
7   echo -n "Who are your best friends? "
8   read -a friends
9   echo "Say hi to ${friends[2]}."
--------------------------------------------------------
```

(输出)

```
    $ nosy
    Are you happy? Yes
1   Yes is the right response.
2   What is your full name? Jon Jake Jones
    Hello Jon
3   Where do you work? the Chico Nut Factory
4   I guess the Chico Nut Factory keeps you busy!
5   Enter your job title: Accountant
6   I thought you might be an Accountant.
7,8  Who are your best friends? Melvin Tim Ernesto
9   Say hi to Ernesto.
```

说明

1. read 命令接收一行用户输入，将其值赋给变量 answer。

2. read 命令从用户处接收输入，将输入的第一个词赋给变量 first，将第二个词赋给变量 middle，然后将直到行尾的所有剩余单词都赋给变量 last。

④ 2.x 版以前的 bash 中不能实现虚线中所示的命令。

3. 从标准输入读取一行，赋值给内置变量 REPLY。

4. 显示变量 REPLY 的值。

5. 带-p 选项的 read 命令，显示提示“Enter your job title:”，把输入行赋值给特定内置变量 REPLY。

6. 在字符串中显示变量 REPLY 的值。

7. 请求用户输入。

8. 带-a 选项的 read 命令将输入当作一组词组成的数组，数组名为 friends，读入数组的元素是 Melvin、Tim 和 Ernesto。

9. 显示 friends 数组的第 3 个元素。数组下标从 0 开始。

范例 14-5

(脚本)

```
   #!/bin/bash
   # Scriptname: printer_check
   # Script to clear a hung-up printer
1  if [ $LOGNAME != root ]
   then
      echo "Must have root privileges to run this program"
      exit 1
   fi
2  cat << EOF
    Warning: All jobs in the printer queue will be removed.
    Please turn off the printer now. Press return when you
    are ready to continue. Otherwise press Control C.
   EOF
3  read JUNK     # Wait until the user turns off the printer
   echo
4  /etc/rc.d/init.d/lpd stop      # Stop the printer
5  echo -e "\nPlease turn the printer on now."
6  echo "Press Enter to continue"
7  read JUNK                # Stall until the user turns the printer back on
   echo                     # A blank line is printed
8  /etc/rc.d/init.d/lpd start     # Start the printer
```

说明

1. 检查用户是否为 root。如不是，则发送错误信息并退出。

2. 创建 here 文档。在屏幕上显示警告信息。

3. read 命令等待用户输入。当用户按下回车键时，变量 JUNK 接收用户之前键入的内容。这个变量没什么实际用处。此处的 read 用来等待用户关闭打印机，然后回来按下回车键。

4. lpd 命令终止打印机后台进程。

5. 现在重新打开打印机。

6. 如果已准备好，请求用户按下回车键。

7. 用户键入的任何内容都被读入变量 JUNK，用户按下回车键之后，程序恢复运行。

8. lpd 程序启动打印机后台进程。



14.3 算术运算

14.3.1 整数运算(declare 和 let 命令)

declare 命令 可以用 declare –i 命令定义整型变量。如果给整型变量赋一个字符串值，则 bash 将把变量赋值为 0。可以对已定义的整型变量执行算术运算(如果变量未被定义为整型变量，内置的 let 命令也允许算术操作。见本节 let 命令部分)。如果给整型变量赋一个浮点数值，则 bash 将报告语法错误。数字可以用不同基数的数字表示，如二进制数，八进制数或十六进制数。

范例 14-6

```
1   $ declare -i num
2   $ num=hello
    $ echo $num
    0
3   $ num=5 + 5
    bash: +: command not found
4   $ num=5+5
    $ echo $num
    10
5   $ num=4*6
    $ echo $num
    24
6   $ num="4 * 6"
    $ echo $num
    24
7   $ num=6.5
    bash: num: 6.5: syntax error in expression (remainder of expression
    is ".5")
```

说明

1. 带-i 选项的 declare 命令创建一个整型变量 num。
2. 试着将字符串“hello”赋值给变量 num，结果变量的值为 0。
3. 如果没有使用 let 命令，就必须把空白符放在引号中或者删掉。
4. 删掉空白符后，运算正确执行。
5. 执行乘法运算，并把结果赋给变量 num。
6. 把空白符放在引号中，则乘法运算可以执行，并阻止 shell 展开通配符(*)。
7. 由于变量设置为整型，赋值时给了一个带小数点的数，导致 bash 报告语法错误。

列出整型变量 只带一个-i 参数的 declare 命令将列出所有已设置的整型变量及其值，如下所示：

```
$ declare -i
declare -ir EUID="15"     # effective user id
declare -ir PPID="235"    # parent process id
declare -ir UID="15"      # user id
```

用不同的基数表示数字 数字可以用不同基数形式表示，包括十进制(默认时基数为 10)，八进制(基数为 8)，十六进制(基数为 16)，另外，基数可以是 2~36 之间的任意整数。

格式

```
variable=base#number-in-that-base
```

范例 14-7

```
n=2#101     # Base is 2; number 101 is in base 2
```

范例 14-8

(命令行)

```
1  $ declare -i x=017
   $ echo $x
   15
2  $ x=2#101
   $ echo $x
   5
3  $ x=8#17
   $ echo $x
   15
4  $ x=16#b
   $ echo $x
   11
```

说明

1. declare 命令用来为整型变量 x 赋一个八进制的值 017。八进制数字必须以一个 0 开头。本例中输出八进制数 017 的十进制值：15。

2. 变量 x 被赋值为二进制数 101，2 代表基数，用#隔开，后面是二进制的数值 101。x 的十进制值为 5。

3. 变量 x 赋值为八进制值 017，x 的十进制值为 15。

4. 变量 x 赋值为十六进制值 b，x 的十进制值为 11。

let 命令 let 命令是 bash shell 内置命令，用来执行整型算术运算和数值表达式测试。可用命令 help let 查看当前 bash 版本支持的 let 操作符，参见 14.5.2 一节中的表 14-4。

范例 14-9

(命令行)

```
1  $ i=5  or let i=5
2  $ let i=i+1
   $ echo $i
   6
3  $ let "i = i + 2"
   $ echo $i
   8
4  $ let "i+=1"
   $ echo $i
   9
5  $ i=3
```



```
6   $ (( i+=4))
    $ echo $i
    7
7   $ (( i=i-2 ))
    $ echo $i
    5
```

说明

1. 变量 i 被赋值为 5。
2. let 命令给变量 i 加 1。在执行算术运算时，不需要用美元符来展开变量。
3. 如果参数包括空白符，则需要使用引号。
4. 简写的操作符+=，用来给变量 i 加 1。
5. 变量 i 被赋值为 5。
6. 双括号可以用来代替 let 命令[⑤]。变量 i 的值加 4。
7. i 的值减 2。也可以写作 i-=2。

14.3.2 浮点数运算

bash 只支持整型运算，但可以使用 bc，awk 和 nawk 工具来处理更复杂的运算。

范例 14-10

```
(命令行)
1   $ n=`echo "scale=3; 13 / 2" | bc`
    $ echo $n
    6.500
2   product=`gawk -v x=2.45 -v y=3.123 'BEGIN{printf "%.2f\n",x*y}'`
    $ echo $product
    7.65
```

说明

1. echo 命令的输出通过管道传送给 bc 程序。变量 scale 赋值为 3，表示小数点后的有效位数为 3。计算 13 除以 2 的值，整个管道用反引号括起来。第二行中将执行命令替换，输出的值赋给变量 n。

2. 通过命令行传递参数列表，gawk 从该列表中获取参数值：x=2.45，y=3.123。乘法运算完成后，printf 函数格式化并显示运算结果，保留小数点后两位数，并将输出赋给变量 product。

14.4 位置参量和命令行参数

14.4.1 位置参量

用户可以通过命令行向脚本传递信息。跟在脚本名后的用空白符分隔的每个词都被称

⑤ 该功能只在 bash 2.x 版本中提供。

为参数。

我们可以在脚本中使用位置参量来引用命令行参数，例如，$1 代表第 1 个参数，$2 代表第 2 个参数，$3 代表第 3 个参数，以此类推。$9 以后，要使用花括号把数字括起来，以表示一个两位数，例如，第 10 个位置参量以${10}的方式来访问。变量$#可以被用来表示参量的个数，而$*则代表所有的参量。我们可以用 set 命令来设置或重设位置参量。如果使用了 set 命令，之前设置的所有位置参量都会被清空。参见表 14-2。

表 14-2　位置参量

位置参量	指代对象
$0	脚本名
$#	位置参量个数
$*	所有的位置参量
$@	未加双引号时，与$*的含义相同
"$*"	单个变量(例如："$1 $2 $3")
"$@"	多个单独的变量(例如："$1" "$2" "$3")
$1 ... ${10}	单独的位置参量

范例 14-11

(脚本)

```
    #!/bin/bash
    # Scriptname: greetings2
    echo "This script is called $0."
1   echo "$0  $1 and $2"
    echo "The number of positional parameters is $#"
    ------------------------------------------------------------
```

(命令行)

```
    $ chmod +x greetings2
2   $ greetings2
    This script is called  greetings2.
    greetings  and
    The number of positional parameters is
3   $ greetings2 Tommy
    This script is called greetings2.
    greetings Tommy and
    The number of positional parameters is 1
4   $ greetings2 Tommy Kimberly
    This script is called greetings2.
    greetings Tommy and Kimberly
    The number of positional parameters is 2
```

说明

1. 在脚本 greetings2 中，位置参量$0 指代脚本名，$1 指代第 1 个命令行参数，$2 则指代第 2 个命令行参数。



2. 不带任何参数执行脚本 greetings2。输出结果说明：脚本名为 greetings2(脚本中的$0)，$1 和$2 没有被赋值，所以它们的值为空，不显示内容。

3. 这次传递了一个参数：Tommy。Tommy 被赋给第一个位置参量。

4. 传入两个参数：Tommy 和 Kimberly。Tommy 被赋给$1，Kimberly 则被赋给$2。

14.4.2 set 命令与位置参量

带参数的 set 命令将重置位置参量[⑥]。位置参量一旦被重置，原来的参量列表就丢失了。要想清除所有的位置参量，可使用 set --命令。$0 始终指代脚本名。

范例 14-12

(脚本)

```
    # !/bin/bash
    # Scriptname: args
    # Script to test command-line arguments
1   echo The name of this script is $0.
2   echo The arguments are $*.
3   echo The first argument is $1.
4   echo The second argument is $2.
5   echo The number of arguments is $#.
6   oldargs=$*
7   set Jake Nicky Scott    # Reset the positional parameters
8   echo All the positional parameters are $*.
9   echo The number of positional parameters is $#.
10  echo "Good-bye for now, $1."
11  set $(date)             # Reset the positional parameters
12  echo The date is $2 $3, $6.
13  echo "The value of \$oldargs is $oldargs."
14  set $oldargs
15  echo $1 $2 $3
```

(输出)

```
    $ args a b c d
1   The name of this script is args.
2   The arguments are a b c d.
3   The first argument is a.
4   The second argument is b.
5   The number of arguments is 4.
8   All the positional parameters are Jake Nicky Scott.
9   The number of positional parameters is 3.
10  Good-bye for now, Jake.
12  The date is Mar 25, 2004.
13  The value of $oldargs is a b c d.
15  Wed Mar 25
```

说明

1. 脚本的名称保存在变量$0 中。

---

⑥ 不带参数时，set 命令将显示该 shell 设置的所有变量，局部变量和导出变量都包括在内。带选项时，set 命令可以打开或关闭 shell 的控制选项，例如-x 和-v。

2. $*代表所有的位置参量。

3. $1 代表第 1 个位置参量(命令行参数)。

4. $2 代表第 2 个位置参量。

5. $#是位置参量(命令行参数)的总个数。

6. 把所有的位置参量都保存在变量 oldargs 中。

7. set 命令重置位置参量，清空原来的位置参量列表。现在，$1 是 Jake，$2 是 Nicky，$3 则是 Scott。

8. $*代表所有位置参量，即 Jake，Nicky 和 Scott。

9. $#代表位置参量的个数，即 3。

10. $1 是 Jake。

11. 执行命令替换之后，即执行 date 命令后，位置参量被重置为 date 命令的输出。

12. 显示$2、$3 和$6 的新值。

13. 显示保存在 oldargs 中的值。

14. set 命令根据 oldargs 中保存的值来创建位置参量。

15. 显示前 3 个位置参量。

范例 14-13

(脚本)

```
    #!/bin/bash
    # Scriptname: checker
    # Script to demonstrate the use of special variable modifiers and
      arguments
1   name=${1:?"requires an argument" }
    echo Hello $name
```

(命令行)

```
2   $ checker
    checker: 1: requires an argument
3   $ checker Sue
    Hello Sue
```

说明

1. 特殊变量修饰符“:?”将检查$1 是否有值。如果$1 无值，则显示指定信息并退出。

2. 不带参数执行该程序。$1 没有被赋值，程序显示报错信息。

3. 给 checker 程序一个命令行参数 Sue。在脚本中，$1 被赋值为 Sue。程序继续运行。

$*和$@的区别 $*和$@仅在被双引号括起来时有区别。$*被括在双引号中时，位置参量列表就变成单个字符串。而$@被括在双引号中时，每个位置参量都被加上引号，也就是说，每个词都被视作一个单独的字符串。

范例 14-14

```
1   $ set 'apple pie' pears peaches
2   $ for i in $*
    > do
    > echo $i
    > done
```




```
    apple
    pie
    pears
    peaches

3   $ set 'apple pie' pears peaches
4   $ for i in "$*"
    > do
    > echo $i
    > done
    apple pie pears peaches

5   $ set 'apple pie' pears peaches
6   $ for i in $@
    > do
    > echo $i
    > done
    apple
    pie
    pears
    peaches

7   $ set 'apple pie' pears peaches
8   $ for i in "$@"                       # At last!!
    > do
    > echo $i
    > done
    apple pie
    pears
    peaches
```

说明

1. 设置位置参量。

2. $*展开后，apple pie 两端的引号被去掉，apple 和 pie 变成了两个单独的词。for 循环将每个词依次赋给变量 i，然后显示 i 的值。每执行一次循环，都将左端的词移走，将下一个词赋给变量 i。

3. 设置位置参量。

4. 给$*加上双引号后，整个参量列表就变成了一个字符串，即"apple pie pears peaches"。整个列表作为单独一个词被赋给 i。循环只执行一次。

5. 设置位置变量。

6. 没有加引号时，$@和$*的用法相同(见上面的第 2 条注释)。

7. 设置位置参量。

8. 给$@加上引号后，每个位置参量都被看作一个加引号的字符串。列表将变成"apple pie"、"peares"、"peaches"，从而实现理想的结果。

14.5 条件结构和流程控制

14.5.1 退出状态

条件结构能够根据某个特定条件是否满足，来选择执行相应的任务。if 命令是最简单的决策形式。if/else 命令提供双路决策，而 if/elif/else 命令则提供多路决策。

bash 可以测试两种类型的条件：命令成功或失败，表达式为真或假。在任何一种类型的测试中，都要使用退出状态。退出状态 0 表示命令成功或表达式为真，非 0 表示命令失败或表达式为假。状态变量“？”中保存的是退出状态值。要查看内存中退出状态是如何起作用的，参见范例 14-15。

范例 14-15

```
   (命令行)
1    $ name=Tom
2    $ grep "$name"  /etc/passwd
     Tom:8ZKX2F:5102:40:Tom Savage:/home/tom:/bin/sh
3    $ echo $?
     0             # Success!
4    $ name=Fred
5    $ grep "$name" /etc/passwd
     $ echo $?
     1             # Failure
```

说明

1. 变量 name 被赋值为字符串 Tom。

2. grep 命令在文件 passwd 中查找字符串 Tom。

3. 变量“？”包含 shell 执行的上一条命令的退出状态，本例中是 grep 命令的退出状态。如果 grep 成功地找到了字符串 Tom，它返回退出状态 0。这里 grep 命令执行成功。

4. 变量 name 被赋值为字符串 Fred。

5. grep 命令在文件 passwd 中查找字符串 Fred，找不到。变量“？”的值为 1，表示 grep 命令失败。

14.5.2 内置命令 test 与 let

单方括号的 test 命令 通常用内置的 test 命令来测试表达式的值，test 命令也被链接到方括号上。这样，既可以使用单独的 test 命令，也可以通过把表达式用单方括号括起来，来测试表达式的值。在用 test 命令或方括号测试表达式时，表达式中的 shell 元字符不会被扩展。由于要对变量进行单词分离，因此包含空白符的字符串必须用引号括起来(参见范例 14-16)。

双方括号的 test 命令 2.x 版 bash 中，用双方括号 [[ ]](内置的 test 复合命令)来测试表达式的值，其中，对变量不进行单词分离，但可以通过元字符扩展进行模式匹配。包含空白符的字符串必须用引号括起来。如果一个字符串(不管含不含空白符)仅仅是在表达式中作为一个普通字符串，而不是一个模式的一部分，则它也必须用引号括起来。逻辑操作



符&&(与)和||(或)代替了与 test 命令一起使用的-a 和-o 选项(参见范例 14-17)。

范例 14-16

```
(test 命令)
(命令行)
1   $ name=Tom
2   $ grep "$name" /etc/passwd
3   $ echo $?
4   $ test $name != Tom
5   $ echo $?
    1            # Failure
6   $ [ $name = Tom ]      # Brackets replace the test command
7   $ echo $?
    0
8   $ [ $name = [Tt]?? ]
    $ echo $?
    1
9   $ x=5
    $ y=20
10  $ [ $x -gt $y ]
    $ echo $?
    1
11  $ [ $x -le $y ]
    $ echo $?
    0
```

说明

1. 变量 name 被赋值为字符串 Tom。

2. grep 命令在文件 passwd 中查找字符串 Tom。

3. 变量“？”包含 shell 执行的上一条命令的退出状态，本例中是 grep 命令的退出状态。如果 grep 成功地找到了字符串 Tom，就返回退出状态 0。这条 grep 命令执行成功。

4. test 命令可用于测试字符串和数字，也可用来执行文件测试。和所有的命令一样，test 也会返回一个退出状态：退出状态为 0，则表达式为真，退出状态为 1，则表达式为假。表达式的等号两侧必须有空格。这条命令测试 name 的值是否等于 Tom。

5. 测试失败，test 返回的退出状态为 1。

6. 方括号是 test 命令的另一种表示方式。第一个方括号后面必须跟空格。这里的表达式测试变量 name 的值是否为字符串 Tom。Bash 中既可以使用单等号来测试字符串是否相等，也可以使用双等号来进行测试。

7. test 的返回值是 0。因为变量 name 的值等于 Tom，所以 test 执行成功。

8. test 命令不允许通配符展开。由于问号被当作一个普通字符，因此 test 执行失败，Tom 与[Tt]??不相等。返回状态为 1，表示该表达式测试失败。

9. 给 x 和 y 赋数值。

10. test 命令使用数值关系操作符进行测试。本例中，test 命令检测$x 是否大于$y，如果是则返回 0，否则，返回 1(见表 14-3)。

11. test 命令检测$x 是否小于或等于$y，如果是则返回 0，否则，返回 1。

表 14-3　test 命令操作符

操 作 符	测 试 内 容
字符串测试	
[string1 = string2]	string1 等于 string2(=两侧必须有空格)
[string1= = string2]	string1 等于 string2(在 2.x 版 bash 中可以用单等号=代替)
[string1 != string2]	string1 不等于 string2(!=两侧必须有空格)
[string]	string 不为空
[-z string]	string 的长度为 0
[-n string]	string 的长度不为 0
[-l string]	string 的长度(字符数) 例如：　test –n $word　或 [-n $word] 　　　　test tom = sue 或 [tom = sun]
逻辑测试	
[string1 –a string2]	string1 和 string2 都为真
[string1 –o string2]	string1 或 string2 至少有一个为真
[!string1]	字符串不匹配
逻辑测试(复合命令)[⑦]	
[[pattern1 && pattern2]]	pattern1 和 pattern2 都为真
[[pattern1 || pattern2]]	pattern1 或 pattern2 至少有一个为真
[[!pattern1]]	模式不匹配
整数测试	
[int1 –eq int2]	int1 等于 int2
[int1 –ne int2]	int1 不等于 int2
[int1 –gt int2]	int1 大于 int2
[int1 –ge int2]	int1 大于或等于 int2
[int1 –lt int2]	int1 小于 int2
[int1 –le int2]	int1 小于或等于 int2
用于文件测试的二进制操作符	
[file1 –nt file2]	如果文件 file1 比 file2 新则为真(根据修改时间)
[file1 –ot file2]	如果文件 file1 比 file2 老则为真
[file1 –ef file2]	如果文件 file1 和 file2 有相同的设备数或 i 结点数则为真

范例 14-17

```
(复合的 test 命令) (bash2.x)
   $ name=Tom; friend=Joseph
1   $ [[ $name == [Tt]om ]]        # Wildcards allowed
    $ echo $?
    0
2   $ [[ $name == [Tt]om && $friend == "Jose" ]]
```

⑦ 使用复合 test 命令时，模式可以包含通配符。要进行严格的字符串测试，pattern2 必须用引号括起来。



```
    $ echo $?
    1
3   $ shopt -s extglob          # Turns on extended pattern matching
4   $ name=Tommy
5   $ [[ $name == [Tt]o+(m)y ]]
    $ echo $?
    0
```

说明

1. 如果使用复合的 test 命令，在字符串测试中可以使用 shell 元字符。本例中的表达式将进行字符串相等测试，看变量 name 是否匹配于 Tom、tom 或 tommy 等。表达式为真，则退出状态(?)为 0。

2. 逻辑操作符&&(与)和 || (或)可以与复合的 test 命令一起使用。如果是&&操作，两边的表达式必须都为真，如果第一个表达式测试为假，则不再进行进一步测试。而对于||逻辑操作符，两个表达式中有一个为真即可，如果第一个表达式测试为真，则不再进行进一步测试。注意："Jose" 是被引号括起来的，如果不使用引号，则是测试变量 friend 是否包含模式 Jose。Jose 和 Joseph 都满足条件。这里第二个条件非真，表达式测试为假，退出状态为 1。

3. 用内置的 shopt 命令打开扩展的模式匹配开关。

4. 变量 name 被赋值为 Tommy。

5. 测试表达式是否相等，包含模式匹配元字符。测试变量 name 是否匹配于一个字符串：它以字母 T 或 t 开头、后跟字母 o、接着是一个或多个字母 m，最后跟字母 y。

后面的范例将说明如何使用内置的 test 命令、带单方括号的 test 命令，以及带双方括号的复合命令来测试退出状态。

let 命令和带双圆括号的算术运算　虽然 test 命令可以计算算术表达式的值，但读者可能更愿意使用 let 命令，因为 let 命令带有丰富的类 C 操作符(bash 2.x)。let 命令可以将表达式包含在一组圆括号中来表达不同的含义。

不管使用的是 test 命令、复合命令还是 let 命令，表达式的结果都会被测试。返回零表示成功，而返回非零状态表示失败(参见表 14-4)。

表 14-4　let 命令操作符

操　作　符	含　　义
-	负号
+	正号
!	逻辑非
~	按位取反
*	乘法
/	除法
%	余数
+	加法
-	减法

(续表)

操 作 符	含 义
<<	按位左移
>>	按位右移
<= >= < >	关系运算符
== !=	相等、不相等
&	按位与操作
^	按位异或操作
|	按位或操作
&&	逻辑与
||	逻辑或
= *= /= %= += -= <<= >>= &= ^= |=	赋值、快捷赋值运算符

范例 14-18 说明了 let 命令中如何使用双圆括号。

范例 14-18

```
(let 命令) (bash2.x)
(命令行)
1   $ x=2
    $ y=3

2   (( x > 2 ))
    echo $?
    1

3   (( x < 2 ))
    echo $?
    0

4   (( x == 2 && y == 3 ))
    echo $?
    0

5   (( x > 2 || y < 3 ))
    echo $?
    1
```

说明

1. 给 x 和 y 赋值。

2. 双圆括号代替 let 命令来测试数值表达式。如果 x 比 y 大，退出状态为 0。由于条件不满足，因此退出状态为 1。变量 ? 包含上一条执行命令的退出状态，这里是(( ))命令的退出状态。注意，括在(( ))中的变量不需要使用美元符$。

3. 用双圆括号测试表达式。如果 x 小于 2，则返回退出状态 0，否则，返回 1。

4. 测试复合表达式：如果 x 等于 2 且 y 等于 3(即两个表达式都为真)，则退出状态返回 0，否则，返回 1。



5. 测试复合表达式：如果 x 大于 2 或 y 小于 3(即有一个表达式为真)，则退出状态返回 0，否则，返回 1。

14.5.3 if 命令

if 命令是条件结构的最简单形式。跟在 if 结构后面的命令(可以是 bash 内置命令或可执行程序)被执行，并返回其退出状态。程序的退出状态通常由编写该程序的程序员决定。退出状态为 0，表示命令执行成功，shell 将执行关键字 then 后面的语句。在 C shell 中，跟在 if 命令后的表达式是布尔型的表达式，与 C 语言一样。但是在 Bash shell，Bourne shell 和 Korne shell 中，跟在 if 后面的则是一条或一组命令。如果该命令的退出状态为 0，shell 就执行从 then 到 fi 之间的语句块。关键字 fi 结束 if 结构。如果退出状态非 0，说明命令由于某种原因运行失败，shell 忽略关键字 then 后的语句，控制跳到紧跟在 fi 语句后面的那条语句。

知道一条被测试命令的退出状态是很重要的。例如，grep 的退出状态能够准确地告诉你 grep 是否在文件中找到了它所查找的模式，如果查找成功，grep 返回退出状态 0，不成功则返回 1。sed 和 awk 程序也查找模式，但是不论是否找到模式，它们都报告一个成功的返回状态。sed 和 awk 判断成功的标准是语法是否正确，而不是从功能上进行判断。

格式

```
  if  命令
  then
       命令
       命令
  fi

(使用 test 命令测试数字和字符串——老的格式)
  if test 表达式
  then
       命令
  fi
   或
  if  [ 字符串/数字表达式 ]  then
       命令
  fi
(使用 test 命令测试字符串——新格式)
  if [[ 字符串表达式 ]] then
      命令
  fi
(使用 let 命令测试数字——新格式)
  if (( 算术表达式 ))
```

范例 14-19

```
1  if grep "$name" /etc/passwd > /dev/null 2>&1
2  then
       echo Found $name!
3  fi
```

说明

1. grep 命令在数据库/etc/passwd 中查找由参数$name 给出的字符串。标准输出和标准错误输出都被重定向到 UNIX 的位容器/dev/null 中。

2. 如果 grep 命令的退出状态为 0，程序转向 then 语句，执行其后的语句，直到 fi 为止。关键字 then 和 fi 之间的命令采用缩排格式，是一种惯例，以使得程序易于阅读和调试。

3. fi 结束跟在 then 语句之后的命令序列。

范例 14-20

```
1  echo "Are you o.k. (y/n) ?"
   read answer
2  if [ "$answer" = Y -o "$answer" = y ]
   then
      echo "Glad to hear it."
3  fi

4  if [ $answer = Y -o "$answer" = y ]
   [: too many arguments

   -------------------------------------------
5  if [[ $answer == [Yy]* || $answer == Maybe ]]⑧
   then
      echo "Glad to hear it."
   fi

6  shopt -s extglob
7  answer="not really"

8  if [[ $answer = [Nn]o?( way|t really) ]]
   then
      echo "So sorry. "
   fi
```

说明

1. 程序显示问题并要求用户回答。read 命令等候用户的响应。

2. 方括号表示的 test 命令，用来测试表达式。如果表达式为真，该命令返回 0。如果表达式为假，则返回 1。如果求得变量 answer 的值为 Y 或 y，则执行语句 then 后面的命令。(测试表达式时，test 命令不允许使用通配符，并且要求方括号和等号两侧都必须有空格。参见表 14-3)。$answer 要用引号括起来，表示是一个独立的字符串，否则 test 命令将失败。

3. fi 终止第 2 行的 if 块。

4. 如果=操作符前的词多于一个，则 test 命令将失败。例如，如果用户输入“yes，you betcha”，则变量 answer 将等于 3 个词，此时如果不用引号把$answer 括起来将导致 test 命令失败。这里显示出错信息。

5. 复合命令操作符[[ ]]允许在字符串表达式中进行 shell 元字符扩展。变量不需要像老的 test 命令中那样用引号括起来，即使是它含有多个词的时候也不需要。且双等号可以用

---

⑧ 第 5~8 行仅能在 bash 2.x 版本上实现。



来替代单个等号。

6. 内置命令 shopt 设置了 extglob，允许扩展参数。参见 14.11.3 节中表 14-11。

7. 变量 answer 被设置为“not really”。

8. 这里使用了扩展的模式匹配。表达式的含义是：如果变量 answer 的值是这样一个字符串，该字符串以 no 或 No 开头，后跟括号中的 0 个或 1 个字符串，那么表达式为真。该表达式可以是 no，No，no way，not really 或 Not really。

exit 命令和变量“？” exit 命令用于终止脚本并返回命令行，以使脚本在某些情况发生时退出。传给 exit 命令的参数是一个 0~255 之间的整数。如果程序返回退出状态 0，则表示程序成功退出。参数非 0 则表示遇到了某种失败。传给 exit 命令的参数被保存在 shell 的变量“？”中。

范例 14-21

```
(脚本)
$ cat bigfiles
   # Name: bigfiles
   # Purpose: Use the find command to find any files in the root
   # partition that have not been modified within the past n (any
   # number within 30 days) days and are larger than 20 blocks
   # (512-byte blocks)

1  if (( $# != 2 ))⑨          # [ $# -ne 2 ]
   then
      echo "Usage:  $0 mdays size " 1>&2
      exit 1
2  fi
3  if (( $1 < 0 || $1 > 30 ))⑩     # [ $1 -lt 0 -o $1 -gt 30 ]
   then
      echo "mdays is out of range"
      exit 2
4  fi

5  if (( $2 <= 20 ))      # [ $2 -le 20 ]
   then
      echo "size is out of range"
      exit 3
6  fi
7  find / -xdev -mtime $1 -size +$2

(命令行)
$ bigfiles
Usage: bigfiles mdays size

$ echo $?
1
$ bigfiles 400 80
mdays is out of range
```

---

⑨ 在早于 bash 2.x 的版本中没有该功能。在旧版中可以写成 if let $ (($# != 2))。

⑩ 在早于 bash2.x 的版本中没有该功能。在旧版中可以写成 if let $ (($1<0 $!>30))。

```
$ echo $?
2
$ bigfiles 25 2
size is out of range
$ echo $?
3
$ bigfiles 2 25
(find 的输出显示在此处)
```

说明

1. 如果参数的个数不等于 2，则显示报错信息并将其发给标准错误输出，然后以状态 1 退出脚本。内置的 test 命令和 let 命令都可以用来测试算术运算表达式的值。

2. fi 标志着 then 后面的语句块结束。

3. 如果从命令行传入的第一个位置参量的值小于 0 或大于 30，就打印报错信息，并以状态 2 退出。有关算术运算符的内容，请参见前面的表 14-4。

4. fi 终结 if 控制块。

5. 如果从命令行传入的第二个位置参量的值大于或等于 20(一个 512 字节的块)，则显示报错信息，并以状态 3 退出。

6. fi 结束 if 控制块。

7. find 命令从根目录开始搜索。选项-xdev 阻止 find 搜索其他分区；选项-mtime 带一个数字参数，该参数表示自文件最后一次被修改以来的天数；选项-size 也带一个数字参数，它表示以 512 字节的块为单位计算的文件大小。

检查空值　检查变量的值是否为空时，必须用双引号把空值括起来，否则 test 命令就会失败。

范例 14-22

```
(脚本)
1   if [ "$name" = "" ]      # Alternative to  [ ! "$name" ] or [ -z "$name" ]
    then
       echo The name variable is null
    fi

(showmount 系统程序显示所有远程装载的系统)
2   remotes=$(/usr/sbin/showmount)
    if [ "X${remotes}" != "X" ]
    then
       /usr/sbin/wall ${remotes}
                   ...
3   fi
```

说明

1. 如果变量 name 的值为空，则测试结果为真。双引号用来表示空值。

2. showmount 命令列出从指定主机远程装载文件系统的所有客户。这条命令可能会列出一个或多个客户，也可能没有输出。于是变量 remotes 可能被赋值，也可能为空。进行测试时，变量 remotes 前面加了一个字母 X。如果 remotes 的值为空，说明没有远程登录过来的客户，于是 X 等于 X，使得程序从行 3 开始执行。如果变量 remotes 有值，比如主机



名 pluto，则表达式变成 if Xpluto != X，于是执行 wall 命令(向远程终端上的所有用户发送消息)。在表达式中使用 X 的目的，是为了确保在 remotes 值为空的情况下，表达式中运算符 != 的两边有一个占位符。

3. fi 结束 if 控制块。

嵌套 if 命令 如果嵌套使用 if 命令，fi 语句则总是与最近的 if 语句配对。对嵌套的 if 语句进行缩进，将有助于显示 fi 语句与 if 语句的配对情况。

14.5.4 if/else 命令

if/else 命令提供一个二路的决策操作。如果 if 后面的命令失败了，就执行 else 后面的命令。

格式

```
if 命令
then
    命令(命令组)
else
    命令(命令组)
fi
```

范例 14-23

(脚本)
```
   #!/bin/bash
   # Scriptname: grepit
1  if grep "$name" /etc/passwd >& /dev/null; then
2      echo Found $name!
3  else
4      echo "Can't find $name."
       exit 1

5  fi
```

说明

1. grep 命令在 NIS passwd 数据库中查找参数 name 给出的字符串。因为用户不需要看到输出结果，所以标准输出和标准错误输出都被重定向到 UNIX 的位容器/dev/null 中。

2. 如果 grep 命令的退出状态为 0，程序的控制转向 then 语句，并执行其后的语句直至遇到 else。

3. 如果 grep 命令没有在 passwd 数据库中找到$name，就执行 else 语句后的命令。也就是说，只有在 grep 命令的退出状态不为 0 时，才执行 else 块。

4. 如果在 passwd 数据库中未找到$name 的值，就执行 echo 语句，之后程序以状态 1 退出，说明查找失败。

5. fi 结束 if 控制块。

范例 14-24

(脚本)
```
#!/bin/bash
```

```
# Scriptname: idcheck
# purpose:check user id to see if user is root.
# Only root has a uid of 0.
# Format for id output:uid=9496(ellie) gid=40 groups=40
# root's uid=0
1  id=`id | gawk -F'[=(]' '{print $2}'`    # get user id
   echo your user id is: $id
2  if  (( id == 0 ))[a]          # [ $id -eq 0] (See cd file: idcheck2)
 $id -eq 0 ] (See cd file: idcheck2)
   then
3      echo "you are superuser."
4  else
       echo "you are not superuser."
5  fi

(命令行)
6  $ idcheck
   your user id is: 9496
   you are not superuser.
7  $ su
   Password:
8  # idcheck
   your user id is: 0
   you are superuser
```

说明

1. id 命令的输出被管道发送给 gawk 命令。gawk 用等号和左圆括号作为字段分隔符，从 id 命令的输出结果中提取出用户的 ID，把结果赋给变量 id。

2，3，4. 如果变量 id 的值等于 0，则执行第 3 行的命令。如果 ID 不等于 0，则执行 else 后的语句。

5. fi 标志着 if 命令结束。

6. UID 为 9496 的当前用户执行 idcheck 脚本。

7. su 命令将用户切换为 root。

8. 命令提示符#表示超级用户(root)是新的当前用户。root 的 UID 是 0。

14.5.5 if/elif/else 命令

if/elif/else 命令提供多路决策操作。如果 if 后的命令失败了，则测试 elif 后的命令。如果测试为成功，就执行它的 then 语句后面的命令。如果 elif 后面的命令也失败了，就检查下一条 elif 命令。如果所有的 elif 命令都不成功，则执行 else 命令。else 操作块称为默认操作。

格式

```
if 命令
then
    命令(命令组)
elif 命令
then
    命令(命令组)
elif 命令
```




```
then
      命令(命令组)
else
      命令(命令组)
  fi
```

范例 14-25

(脚本)

```
   #!/bin/bash
   # Scriptname: tellme
   # Using the old-style test command

1  echo -n "How old are you? "
   read age
2  if [ $age -lt 0 -o $age -gt 120 ]
   then
      echo  "Welcome to our planet! "
      exit 1
   fi
3  if  [ $age -ge 0 -a $age -le 12 ]
   then
      echo "A child is a garden of verses"
   elif [ $age -gt 12 -a $age -le 19 ]
   then
      echo "Rebel without a cause"
   elif [ $age -gt 19 -a  $age -le 29 ]
   then
      echo "You got the world by the tail!!"
   elif [ $age -gt  29 -a  $age -le 39 ]
   then
      echo "Thirty something..."
4  else
      echo "Sorry I asked"
5  fi
```

(输出)

```
$ tellme
How old are you? 200
Welcome to our planet!

$ tellme
How old are you? 13
Rebel without a cause

$ tellme
How old are you? 55
Sorry I asked
-----------------------------------------------------------
   #!/bin/bash
   # Using the new (( )) compound let command
   # Scriptname: tellme2
```

```
1   echo -n "How old are you? "
    read age
2   if (( age < 0 || age > 120 ))
    then
       echo "Welcome to our planet! "
       exit 1
    fi
3   if ((age >= 0 && age <= 12))
    then
       echo "A child is a garden of verses"
    elif ((age > 12 && age <= 19 ))
    then
       echo "Rebel without a cause"
    elif (( age > 19 &&  age <=  29 ))
    then
       echo "You got the world by the tail!!"
    elif  (( age >  29 &&  age <= 39 ))
    then
       echo "Thirty something..."
4   else
       echo "Sorry I asked"
5   fi
```

说明

1. 请求用户输入，将用户的输入赋给变量 age。

2. 用 test 命令执行数值测试。如果 age 小于 0 或大于 120，就执行 echo 命令，然后程序以状态 1 终止。屏幕上将出现交互式 shell 的提示符。

3. 用 test 命令执行数值测试。如果 age 大于等于 0 并且小于 12，test 命令就返回退出状态 0，即真，并且执行 then 后面的语句。否则，程序控制转到 elif。如果 elif 的测试结果为假，再测试下一个 elif。

4. else 结构是默认操作。如果之前的语句都不为真，则执行 else 命令。

5. fi 结束最外层的 if 语句。

14.5.6 文件测试

编写脚本时常常需要使用某些特定文件，且这些文件需拥有特定的权限、属于某个类型或具有其他一些属性。实际中，对文件进行测试是编写可靠脚本的一个必要组成部分。

表 14-5 文件测试操作符

测试操作符	测试结果为真时需满足的条件
- b filename	块专用文件
- c filename	字符专用文件
- d filename	目录存在
- e filename	文件存在



(续表)

测试操作符	测试结果为真时需满足的条件
-f filename	普通文件存在且不是目录
-G filename	文件存在且属于有效组 ID 时为真
-g filename	Set-group-ID 被设置
-k filename	Sticky 位被设置
-L filename	文件是一个符号链接
-p filename	文件是一个命名管道
-O filename	文件存在且属于有效用户 ID
-r filename	文件可读
-S filename	文件是一个 socket
-s filename	文件大小非 0
-t fd	如果 fd(文件描述符)被一个终端打开则为真
-u filename	Set-user-ID 位被设置
-w filename	文件可写
-x filename	文件可执行

范例 14-26

(脚本)

```
   #!/bin/bash
   # Using the old-style test command [ ] single brackets
   # filename: perm_check
   file=./testing

1  if [ -d $file ]
   then
      echo "$file is a directory"
2  elif [ -f $file ]
   then
3     if [ -r $file -a -w $file -a -x $file ]
      then              # nested if command
         echo "You have read,write,and execute permission on $file."
4     fi
5  else
      echo "$file is neither a file nor a directory. "
6  fi
--------------------------------------------------------------------
   #!/bin/bash
   # Using the new compound operator for test [[ ]] ⑪11
   # filename: perm_check2
   file=./testing
```

⑪ 新式风格的 test 命令，使用了复合方括号符，但在早于 bash 2.x 的版本中不能使用。

```
1  if [[ -d $file ]]
   then
      echo "$file is a directory"
2  elif [[ -f $file ]]
   then
3     if [[ -r $file && -w $file && -x $file ]]
      then    # nested if command
          echo "You have read,write,and execute permission on $file."
4      fi
5  else
      echo "$file is neither a file nor a directory. "
6  fi
```

说明

1. 如果文件 testing 是一个目录，则显示“testing is a directory”。
2. 如果文件 testing 不是目录，文件是普通文件，则继续向下执行。
3. 如果文件 testing 可读、可写而且可执行，则继续向下执行。
4. fi 结束最内层的 if 命令。
5. 如果行 1 和行 2 都不为真，则执行 else 命令。
6. 这个 fi 对应第一个 if。

14.5.7 null 命令

冒号代表的 null 命令是 shell 的一个内置命令，它不做任何工作，只返回退出状态 0。null 命令有时被放在 if 命令后面作为一个占位符，这时 if 命令没什么事可做，却需要有一条命令放在 then 后面，否则，程序会产生报错信息，因为 then 语句后面必须有内容。null 命令常常被用作循环命令的参数，作用是让循环无限执行下去。

范例 14-27

(脚本)

```
   #!/bin/bash
   # filename: name_grep

1  name=Tom
2  if grep "$name" databasefile >& /dev/null
   then
3     :
4  else
      echo "$1 not found in databasefile"
      exit 1
   fi
```

说明

1. 变量 name 被赋值为字符串 Tom。
2. if 命令测试 grep 命令的退出状态。如果在文件 databasefile 中找到了字符串 Tom，就执行 null 命令(即冒号)，不做任何操作。标准输出和错误输出都被重定向到/dev/null。



3. 冒号代表 null 命令。除了返回退出状态 0 外，这条命令不做任何操作。

4. 如果没找到 Tom，就显示一条报错信息并退出。若 grep 命令运行失败，则执行 else 后面的命令。

范例 14-28

（命令行）

```
1   $ DATAFILE=
2   $ : ${DATAFILE:=$HOME/db/datafile}
    $ echo $DATAFILE
    /home/jody/ellie/db/datafile
3   $ : ${DATAFILE:=$HOME/junk}
    $ echo $DATAFILE
    /home/jody/ellie/db/datafile
```

说明

1. 变量 DATAFILE 被赋值为空。

2. 冒号命令是空命令。修饰赋(:=)返回一个可以赋给变量或用于测试的值。本例中，表达式作为参数传给这条空命令。shell 会执行变量替换，即若此时 DATAFILE 尚未赋值，则将路径名赋给它。这样，变量 DATAFILE 就始终都会有值。

3. 由于变量 DATAFILE 已被设置，所以 shell 不会再用修饰符:=右边提供的默认值重置它。

范例 14-29

（脚本）

```
    #!/bin/bash
    # Scriptname: wholenum
    # Purpose:The expr command tests that the user enters an integer

1   echo "Enter an integer."
    read number
2   if expr "$number" + 0 >& /dev/null
    then
3       :
    else
4       echo "You did not enter an integer value."
        exit 1
5   fi
```

说明

1. 请求用户输入一个整数。将该整数赋给变量 number。

2. expr 命令对表达式求值。如果加法能顺利执行，则说明用户输入的确实是一个整数，expr 就返回成功的退出状态。所有输出都被重定向到位容器/dev/null 中。

3. 若 expr 执行成功，则返回退出状态 0，冒号命令不做任何操作。

4. 若 expr 执行失败，则返回非 0 的退出状态，先由 echo 命令显示指定消息，然后程序退出。

5. fi 结束 if 块。

14.5.8 case 命令

case 命令是一个多路分支命令，可用来代替 if/elif 命令。case 变量的值与 value1，value2 等的值逐一比较，直至找到与之匹配的值。如果某个值与 case 变量匹配，程序就执行该值后面的命令，直至遇到双分号，然后跳到词 esac(case 倒过来拼写)后面继续往下执行。

如果没有找到与 case 变量匹配的值，程序就执行默认值“*)”后面的命令，直至遇到“;;”或 esac。值*) 的功能与 if/else 条件命令中的 else 语句相同。case 的表达式里可以用 shell 通配符，还可以用竖杠(管道符)将两个值相或。

格式

```
case 变量 in
值 1)
    命令(命令组)
 ;;
值 2)
    命令(命令组)
 ;;
*)
    命令(命令组)
 ;;
esac
```

范例 14-30

(脚本)

```
    #!/bin/bash
    # Scriptname: xcolors

1   echo -n "Choose a foreground color for your xterm window: "
    read color
2   case "$color" in
3   [Bb]l??)
4        xterm -fg blue -fn terminal &
5        ;;
6   [Gg]ree*)
        xterm -fg darkgreen -fn terminal &
        ;;
7   red | orange)           # The vertical bar means "or"
        xterm -fg "$color"  -fn terminal &
        ;;
8   *)
        xterm -fn terminal
        ;;
9   esac
10  echo  "Out of case command"
```

说明

1. 请求用户输入。将输入保存在变量 color 中。
2. case 命令对表达式$color 求值。



3. 如果变量 color 的值以 B 或 b 开头，后面跟字母 l 和两个任意字符，则 case 表达式匹配第一个值。这里的值以单个圆括号终止，其中的通配符是用于文件名扩展的 shell 元字符。xterm 命令将终端前景色设置为蓝色。

4. 如果第 3 行的值与 case 表达式相匹配，则执行该语句。

5. 在命令块的最后一条命令后，必须用双分号。程序执行到双分号时，就会将控制跳到第 10 行。把双分号单独写一行可以方便脚本的阅读和调试。

6. 如果 case 表达式匹配以 G 或 g 开头，后跟字母 ree，以 0 个或多个任意字符结尾的值，则执行 xterm 命令。双分号结束该语句块，控制跳转到第 10 行。

7. 竖杠用作条件运算符或。如果 case 表达式匹配 red 或 orange，则执行 xterm 命令。

8. 这是默认值。如果以上所有值都不能匹配 case 表达式，就执行“*)”后面的命令。

9. esac 语句结束 case 命令。

10. 当匹配 case 的一个值后，程序由此继续执行。

用 here 文档和 case 命令生成菜单 here 文档经常与 case 命令结合起来使用。我们可以用 here 文档生成一个选项菜单显示在屏幕上，要求用户选择一个菜单项，然后用 case 命令对照选项集测试用户的输入，以执行相应的命令。

范例 14-31

```
(来源于.bash_profile 文件)
    echo "Select a terminal type:  "
1   cat <<- ENDIT
        1) unix
        2) xterm
        3) sun
2   ENDIT
3   read choice
4   case "$choice" in
5   1)   TERM=unix
         export TERM
         ;;
    2)   TERM=xterm
         export TERM
         ;;
6   3)   TERM=sun
         export TERM
         ;;
7   esac
8   echo "TERM is $TERM."

(命令行及输出)
$ . .bash_profile
    Select a terminal type:
    1) unix
    2) xterm
    3) sun
    2                <-- User input
    TERM is xterm.
```

说明

1. 如果把这段脚本放在.bash_profile 中，当您登录成功后，就可以选择不同的终端类型。here 文档被用来显示选项菜单。
2. 用户自定义的终止符 ENDIT 标志 here 文档的结束。
3. read 命令把用户的输入保存到变量 TERM 中。
4. case 命令求出变量 TERM 的值，将其与右圆括号前面的值(1、2 和*)逐一比较。
5. 测试的第一个值是 1。如果二者匹配，就将终端类型设置为 unix。导出变量 TERM，让子 shell 能够继承它。
6. 不需要默认值。通常是在登录时在/etc/profile 中设置变量 TERM。如果用户选择 3，则将终端类型设置为 sun。
7. esac 终结 case 命令。
8. case 命令结束后，执行这一行。

14.6 循环命令

循环命令用于将一个或一组命令执行指定的次数，或者一直执行直到满足某个条件为止。Bash shell 提供了 3 种类型的循环：for 循环、while 循环和 until 循环。

14.6.1 for 命令

for 循环命令用于在某个项目列表上将命令执行指定次数。例如，您可能会用 for 循环在某个文件或用户名列表上重复执行相同的命令。for 命令后面跟一个用户自定义的变量、关键字 in 和一组词。执行第一轮循环时，先将词表中的第一个词赋给变量，并把该词从词表中移走，然后进入循环体，执行关键字 do 和 done 之间的命令。下一次进入循环时，则将第二个词赋给变量，以此类推。循环体从关键字 do 开始，到关键字 done 结束。当词表中所有的词都被移走后，循环就结束了，程序控制从关键字 done 之后继续执行。

格式

```
for 变量 in 词表
do
  命令(命令组)
done
```

范例 14-32

(脚本)

```
   #!/bin/bash
   # Scriptname: forloop
1  for pal in Tom Dick Harry Joe
2  do
3      echo "Hi $pal"
4  done
```



```
5   echo "Out of loop"
```

（输出）
```
Hi Tom
Hi Dick
Hi Harry
Hi Joe
Out of loop
```

说明

1. for 循环将遍历一列组名：Tom、Dick、Harry 和 Joe，用完一个就移走一个(往左移，并且将它的值赋给用户定义的变量 pal)。一旦所有的词都被移走，词表为空，循环就结束了，程序从关键字 done 之后接着往下执行。执行第一轮循环时，变量 pal 将被赋值为 Tom；第二轮循环时，pal 将被赋值为 Dick；再下一轮，pal 将被赋值为 Harry；最后一轮，pal 将被赋值为 Joe。

2. 词表后面必须跟关键字 do。如果把 do 和词表放在同一行，就必须用分号隔开。例如：for pal in Tom Dick Harry Joe；do。

3. 这是循环体。程序把 Tom 赋给变量 pal 之后，就执行循环体中的命令(即关键字 do 和 done 之间的所有命令)。

4. 关键字 done 结束循环。一旦词表中最后一个词(Joe)被赋给变量并移开，循环将退出。

5. 退出循环后，控制由此重新开始。

范例 14-33

（命令行）
```
1   $ cat mylist
    tom
    patty
    ann
    jake
```

（脚本）
```
    #!/bin/bash
    # Scriptname: mailer
2   for person in $(cat mylist)
                  # `cat mylist` command substitution the alternate way
    do
3       mail $person < letter
        echo $person was sent a letter.
4   done
5   echo "The letter has been sent."
```

说明

1. 显示文件 mylist 的内容。

2. 执行命令替换，文件 mylist 的内容变成了一个词表。执行第一轮循环时，tom 被赋给变量 person，然后被移开，patty 顶替它的位置，之后各轮循环以此类推。

3. 在循环体中，向每个用户邮寄一份 letter 文件。

4. 关键字 done 标志该轮循环的结束。

5. 给表中所有用户都发送邮件之后，程序退出循环，接着执行这一行。

范例 14-34

（脚本）

```
    #!/bin/bash
    # Scriptname: backup
    # Purpose: Create backup files and store
    # them in a backup directory.
    #

1   dir=/home/jody/ellie/backupscripts
2   for file in memo[1-5]
    do
3       if [ -f $file ]
        then
            cp $file $dir/$file.bak
            echo "$file is backed up in $dir"
        fi
4   done
```

（输出）

```
memo1 is backed up in /home/jody/ellie/backupscripts
memo2 is backed up in /home/jody/ellie/backupscripts
memo3 is backed up in /home/jody/ellie/backupscripts
memo4 is backed up in /home/jody/ellie/backupscripts
memo5 is backed up in /home/jody/ellie/backupscripts
```

说明

1. 将变量 dir 赋值为保存备份脚本的路径。

2. 词表由当前工作目录中所有名字以 memo 开头、以 1~5 之间的数字结尾的文件组成。各轮循环将文件名逐个赋给变量 file。

3. 进入循环体后，程序将对文件进行检查，确保其存在并且是一个真正的文件。如果确实如此，就将它的文件名加上后缀.bak 并复制到目录/home/jody/ellie/backupscripts 中。

4. done 结束循环。

14.6.2 词表中的$*和$@变量

$*和$@扩展的结果几乎完全一样，唯一不同的是当它们被括在双引号中时，$*的值是一个字符串，而$@的值则是一组相互独立的词。

范例 14-35

（脚本）

```
    #!/bin/bash
    # Scriptname: greet
1   for name in $*        # same as for name in $@
2   do
        echo Hi $name
```




```
3    done

(命令行)
$ greet Dee Bert Lizzy Tommy
Hi Dee
Hi Bert
Hi Lizzy
Hi Tommy
```

说明

1. $*和$@被展开后是一个所有位置参量的列表，本例中，它们被展开后的结果就等于从命令行传入的参数：Dee、Bert、Lizzy 和 Tommy。列表中的每个名字被依次赋给 for 循环的变量 name。

2. 执行循环体中的命令，直到列表为空。

3. 关键字 done 标志循环体的结束。

范例 14-36

```
(脚本)
    #!/bin/bash
    # Scriptname: permx

1   for file        # Empty wordlist
    do
2     if [[ -f $file && ! -x $file ]]
      then
3           chmod +x $file
            echo $file now has execute permission
      fi
    done
(命令行)
4   $ permx *
   addon now has execute permission
   checkon now has execute permission
   doit now has execute permission
```

说明

1. 如果没有为 for 循环提供参数列表，它就对所有位置参量进行遍历。这行等同于 for file in $*命令。

2. 文件名将来自命令行。shell 将星号(*)扩展为当前工作目录中所有的文件名。如果该文件是一个没有执行权限的文本文件，就执行 3 的命令。

3. 给每个被处理的文件加上执行权限。

4. shell 把命令行里的星号作为通配符进行求值，将它替换为当前目录下的所有文件。shell 把这些文件作为参数传给脚本 permx。

14.6.3 while 命令

while 命令对紧跟在它后面的命令进行测试，如果该命令的退出状态为 0，就执行循环

体内的命令(即关键字 do 和 done 之间的命令)。执行到关键字 done 后，控制回到循环的顶部，while 命令再次检查该命令的退出状态。循环将一直继续下去，直到该命令的退出状态非 0 为止。该命令的退出状态非 0 时，程序将从关键字 done 之后开始执行。

格式

```
while 命令
do
  命令(命令组)
done
```

范例 14-37

(脚本)

```
    #!/bin/bash
    # Scriptname: num
1   num=0           # Initialize num
2   while (( $num < 10 ))⑫ # or while [ num -lt 10 ]
    do
        echo -n "$num "
3       let num+=1          # Increment num
    done
4   echo -e "\nAfter loop exits, continue running here"
```

(输出)

```
    0 1 2 3 4 5 6 7 8 9
4   After loop exits, continue running here
```

说明

1. 首先初始化，将变量 num 设为 0。

2. while 命令后是一条 let 命令。let 命令对算术表达式进行测试，当条件为真时则返回一个退出状态 0(即真)，也就是说，当 num 的值小于 10 时就进入循环体。

3. 在循环体中，num 的值加 1。如果 num 的值始终不变，循环就会无休止地重复下去，直至该进程被终止。

4. 循环退出后，echo 命令(带-e 选项)显示一个换行符和字符串。

范例 14-38

(脚本)

```
    #!/bin/bash
    # Scriptname: quiz
1   echo "Who was the 2nd U.S. president to be impeached?"
    read answer
2   while [[ "$answer" != "Bill Clinton" ]]
3   do
        echo "Wrong try again!"
4       read answer
5   done
6   echo You got it!
```

⑫ bash 2.x 版中使用的形式。



(输出)
```
Who was the 2nd U.S. president to be impeached? Ronald Reagan
Wrong try again!
Who was the 2nd U.S. president to be impeached?  I give up
Wrong try again!
Who was the 2nd U.S. president to be impeached?  Bill Clinton
You got it!
```

说明

1. echo 命令向用户提问：Who was the 2nd president to be impeached(第二个被弹劾的美国总统是谁)？read 命令等待用户输入，用户的输入将被保存在变量 answer 中。

2. 进入 while 循环，test 命令(即方括号)测试该表达式。如果变量 answer 的值不等于字符串 Bill Clinton，就进入循环体，执行关键字 do 和 done 之间的命令。

3. 关键字 do 是循环体的开始。

4. 要求用户重新输入答案。

5. 关键字 done 标志着循环体结束。控制返回 while 循环的顶部，再次对表达式进行测试。只要变量 answer 的值不等于 Bill Clinton，循环就会不停的重复执行下去。一旦用户输入 Bill Clinton，循环就结束。程序控制转到标为 6 的那一行。

6. 完成循环体的执行后，控制由此开始。

范例 14-39

(脚本)
```
$ cat sayit
   #!/bin/bash
   # Scriptname: sayit
   echo  Type q to quit.
   go=start
1  while [[ -n "$go" ]]   # Make sure to double quote the variable
   do
2      echo -n I love you.
3      read word
4      if [[ $word == [Qq] ]]
       then        # [ "$word" = q -o "$word" = Q ]  Old style
          echo "I'll always love you!"
          go=
       fi
   done
```

(输出)
```
Type q to quit.
I love you.       <-- When user presses Enter, the program continues
I love you.
I love you.
I love you.
I love you.q
```

```
I'll always love you!
$
```

说明

1. 执行 while 后面那条命令，并测试它的退出状态。test 命令的选项-n 测试字符串是否非空。由于变量 go 有初值，所以测试成功，test 返回退出状态 0。如果变量 go 没有加双引号，值又为空，test 命令就会显示信息：

```
go: test: argument expected
```

2. 进入循环。在屏幕上显示字符串“I love you.”。

3. read 命令等待用户输入。

4. 测试这个表达式。如果用户输入字母 q 或 Q，就显示字符串“I'll always love you!”，并且将变量 go 设为空。当再次进入 while 循环时，由于变量 go 为空，所以测试不成功，循环终止。控制转到 done 语句后面的行。本例中，因为 done 后面没有其他要执行的行，所以脚本将终止。

14.6.4 until 命令

until 命令的用法与 while 命令类似，不过 until 命令只在它后面的命令失败时(即命令返回非 0 的退出状态时)，才执行循环语句。执行到关键字 done 时，控制回到循环顶部，until 命令再次检查它后面命令的退出状态，循环将继续执行，直到 until 测试到那条命令的退出状态变成 0 为止。当该命令的退出状态为 0 时，循环退出，程序从关键字 done 的下一行开始执行。

格式

```
until 命令
do
  命令(或命令组)
done
```

范例 14-40

```
#!/bin/bash
1   until who | grep linda
2   do
        sleep 5
3   done
    talk linda@dragonwings
```

说明

1. until 循环测试管道中最后一条命令 grep 的退出状态。who 命令列出当前有哪些用户登录在这台机器上，并将它的输出通过管道传给 grep。只有当 grep 命令在 who 命令的输出中找到用户 linda 时，grep 命令才返回退出状态 0(命令执行成功)。

2. 如果用户 linda 还没有登录，就进入循环体，程序暂停 5 秒。

3. 用户 linda 登录后，grep 命令的退出状态将变成 0，控制也将转到关键字 done 下面那条语句。



范例 14-41

```
(脚本)
    #!/bin/bash
    # Scriptname: hour

1   hour=0
2   until (( hour > 24 ))
    do
3       case "$hour" in
        [0-9]|1[0-1]) echo "Good morning!"
            ;;
        12)  echo "Lunch time."
            ;;
        1[3-7])echo "Siesta time."
            ;;
        *)   echo "Good night."
            ;;
        esac
4       (( hour+=1 ))   # Don't forget to increment the hour
5   done
(输出)
Good morning!
Good morning!
    ...
Lunch time.
Siesta time.
    ...
Good night.
    ...
```

说明

1. 变量 hour 被初始化为 0。

2. let 命令测试 hour 的值是否大于 24。如果 hour 的值不大于 24，则进入循环体。until 后面那条命令返回一个非 0 的退出状态时，程序就进入 until 循环。循环将反复执行，直到该条件为真。

3. case 命令计算变量 hour 的值，并测试每条 case 语句以寻找与 hour 匹配的值。

4. 控制返回循环顶部之前，变量 hour 的值加 1。

5. done 命令标志循环体的结束。

14.6.5 select 命令和菜单

here 文档是生成菜单的简便方法，而 bash 提供了另一种循环机制，称为 select 循环，它主要用于创建菜单。 按数字顺序排列的菜单项将列表显示在标准错误输出上，并显示 PS3 提示符请求用户输入(默认时，PS3 值为“#?”)。显示 PS3 提示符后，shell 等待用户输入，输入的应当是菜单列表中的一个数字。输入值保存在一个 shell 的特殊变量 REPLY 中，它与选项列表中相应行的括号右面的字符串相关联。

case 命令和 select 命令联合使用时，用户可以从菜单中进行选择，并基于选项执行相应的命令。LINES 和 COLUMNS 变量，用来确定菜单在终端上的布局(这两个变量是 2.x 版 bash 的内置变量，但在此之前的 bash 版本中没有。如果您所使用的 bash shell 中它们还没有定义，可以在.bash_profile 文件中定义并导出它们)。输出被显示在标准错误上，每一项的开头是一个数字和右括号，PS3 提示符显示在菜单底部。因为 select 命令是一个循环命令，因此，一定要记住用 break 命令退出循环，或者用 exit 命令退出脚本程序。

格式

```
select var in wordlist
do
命令(或命令组)
done
```

范例 14-42

(脚本)

```
    #!/bin/bash
    # Scriptname: runit

1   PS3="Select a program to execute: "
2   select program in 'ls -F' pwd date
3   do
4           $program
5   done
```

(命令行)

```
Select a program to execute: 2
1) ls -F
2) pwd
3) date
/home/ellie
Select a program to execute: 1
1) ls -F
2) pwd
3) date
12abcrty abc12  doit* progs/  xyz
Select a program to execute: 3
1) ls -F
2) pwd
3) date
Sun Mar 12 13:28:25 PST 2004
```

说明

1. PS3 变量被赋值为提示语句，出现在菜单选项的下面。而默认的 PS3 提示符为“$#”，并送到标准错误输出上，即屏幕上。

2. select 循环由 program 变量和显示在菜单上的词列表(ls －F，pwd 和 date)组成。这里列表中的词都是 UNIX/Linux 命令，当然它们也可以是任何其他的词语，如 red，green，yellow，或是 cheese，bread，milk，crackers 等。如果词语中有空格，就要用引号把词括起



来，如 ' ls -F'。

3. do 关键字表示 select 循环开始。

4. 用户在在菜单中选择数字后，相当于选择了括号右边的词语的值，例如，如果选择了数字 2，2 与词 pwd 关联，那么 pwd 将被赋值给变量 program。$program 解释为命令 pwd，并执行该命令。

5. done 命令标志着 select 循环体中的语句结束，控制返回到循环顶部，循环将一直执行，直到用户按下 Ctrl+C 组合键。

范例 14-43

（脚本）

```
    #!/bin/bash
    # Scriptname: goodboys

1   PS3="Please choose one of the three boys : "
2   select choice in tom dan guy
3   do
4       case "$choice" in
        tom)
            echo Tom is a cool dude!
5           break;;         # break out of the select loop
6       dan | guy )
            echo Dan and Guy are both wonderful.
            break;;
        *)
7           echo "$REPLY is not one of your choices" 1>&2
            echo "Try again."
            ;;
8       esac
9   done
```

（命令行）

```
$ goodboys
1) tom
2) dan
3) guy
Please choose one of the three boys : 2
Dan and Guy are both wonderful.
$ goodboys
1) tom
2) dan
3) guy
Please choose one of the three boys : 4
4 is not one of your choices
Try again.
Please choose one of the three boys : 1
Tom is a cool dude!
$
```

说明

1. PS3 提示符将打印在菜单的下面。
2. 进入 select 循环，列表中的词显示为一个按数字排序的菜单。
3. 循环体开始。
4. 变量 choice 被赋值为列表中的第一个值，然后该值从列表中移出，下一项将为第一个值。
5. break 语句把循环控制跳转到第 9 行。
6. 如果选择了 guy 或 dan，则执行后面的 echo 命令，echo 命令后是 break 命令，把控制跳转到第 9 行。
7. 内置的 REPLY 变量中保存当前列表中选项的序号，如 1，2 或 3。
8. esac 标志着 case 命令的结束。
9. done 标志着 select 循环的结束。

范例 14-44

（脚本）

```
    #!/bin/bash
    # Scriptname: ttype
    # Purpose: set the terminal type
    # Author: Andy Admin

1   COLUMNS=60
2   LINES=1
3   PS3="Please enter the terminal type: "
4   select choice in wyse50 vt200 xterm sun
    do
5       case "$REPLY" in
        1)
6           export TERM=$choice
            echo "TERM=$choice"
            break;;          # break out of the select loop
        2 | 3 )
            export TERM=$choice

            echo "TERM=$choice"
            break;;
        4)
            export TERM=$choice
            echo "TERM=$choice"
            break;;
        *)
7           echo -e "$REPLY is not a valid choice. Try again\n" 1>&2
8           REPLY=    # Causes the menu to be redisplayed
                ;;
        esac
9   done
```



```
(命令行)
$ ttype
1) wyse50    2) vt200    3) xterm    4) sun
Please enter the terminal type : 4
TERM=sun

$ ttype
1) wyse50    2) vt200    3) xterm    4) sun
Please enter the terminal type : 3
TERM=xterm

$ ttype
1) wyse50    2) vt200    3) xterm    4) sun
Please enter the terminal type : 7
7 is not a valid choice. Try again.

1) wyse50    2) vt200    3) xterm    4) sun
Please enter the terminal type: 2
TERM=vt200
```

说明

1. COLUMNS 变量的值被设置为菜单的宽度，即 select 循环生成的菜单在终端上显示的列数。默认值为 80。

2. LINES 变量控制菜单在终端显示垂直方向上的行数，默认为 24 行。如果把 LINES 变量的值置为 1，菜单项将打印在一行上，而不是如上例所示的垂直显示。

3. 设置 PS3 提示符，该提示符的内容将显示在菜单选项下。

4. select 循环打印出一个有 4 个选项的菜单：wyse50，vt200，xterm 和 sun。根据变量 REPLY 中保存的用户响应值，choice 将被赋值为以上所列的某个值。如果 REPLY 为 1，将 wyse50 赋给 choice；如果 REPLY 为 2，将 vt200 赋给 choice；如果 REPLY 是 3，将 xterm 赋给 choice；如果 REPLY 是 4，将 sun 赋给 choice。

5. REPLY 变量等于用户输入的选择。

6. 给终端类型赋值，并导出、显示该变量。

7. 如果用户没有输入 1 和 4 之间的数字，用户将被提示重新输入。注意，此时将仅显示 PS3 提示符，而不显示菜单。

8. 如果将 REPLY 值设为空(null)，如 REPLY= ，则将重新显示菜单。

9. select 循环终止。

14.6.6 循环控制命令

有些情况下，可能需要从循环中跳出来，或返回循环顶部，或者用某种办法中断某个死循环。Bash shell 提供了一组循环控制命令来处理这类情况。

shift 命令　shift 命令将参量列表左移指定次数。没有给定参数时，shift 命令把参量列表左移一次。一旦列表被移动，左端那个参数就从列表中删除了。while 循环遍历位置参量列表时，常常会用到 shift 命令。

格式

```
shift [n]
```

范例 14-45

(没有循环)

(脚本)

```
    #!/bin/bash
    # Scriptname: shifter
1   set joe mary tom sam
2   shift
3   echo $*
4   set $(date)
5   echo  $*
6   shift 5
7   echo  $*
8   shift 2
```

(输出)

```
3   mary tom sam
5   Thu Mar 18 10:00:12 PST 2004
7   2001
8   shift: shift count must be <= $#
```

说明

1. set 命令设置位置参量。$1 被设为 joe，$2 被设为 mary，$3 被设为 tom，$4 则被设为 sam。$*代表所有位置参量。

2. shift 命令把位置参量左移，joe 被移除。

3. 显示被移动之后的参量列表。

4. set 命令把位置参量重置为 UNIX 命令 date 的输出。

5. 显示新的参量列表。

6. 把参量列表左移 5 次。

7. 显示新的参量列表。

8. 若左移的次数超过了列表中的参数个数，shell 会向标准错误输出发送一条消息，声明 shift 命令不能移动比列表中的参数个数更多的次数。$#是位置参量的总个数。在 2.x 版的 bash 中，不显示错误信息。

范例 14-46

(有循环)

(脚本)

```
    #!/bin/bash
    # Name: doit
    # Purpose: shift through command-line arguments
    # Usage: doit [args]
1   while (( $# > 0 ))
    do
```




```
2       echo  $*
3       shift
4   done
```

（命令行与输出）

```
$ doit a b c d e
a b c d e
b c d e
c d e
d e
e
```

说明

1. while 命令测试一个数值表达式。如果位置参量的个数($#)大于 0，就进入循环体。位置参量来自命令行参数。本例中有 5 个位置参量。

2. 显示所有的位置参量。

3. 将参量列表左移一次。

4. 循环体到此结束，控制返回循环顶部。每一次进入循环后，shift 命令都使得参量列表减少一个成员。第一次移动后，$#(位置参量的个数)变成 4。当$#减到 0 时，循环结束。

范例 14-47

（脚本）

```
    #!/bin/bash
    # Scriptname: dater
    # Purpose: set positional parameters with the set command
    # and shift through the parameters.

1   set $(date)
2   while (( $# > 0 ))
    do
3      echo $1
4      shift
    done
```

（输出）

```
Wed
Mar
17
19:25:00
PST
2004
```

说明

1. set 命令取得 date 命令的输出，然后将其赋给从$1~$6 这 6 个位置参量。

2. while 命令测试位置参量的个数是否大于 0。如果是大于 0，就进入循环体。

3. echo 命令显示$1(第一个位置参量)的值。

4. shift 命令把参量列表左移一次。每一轮循环都将列表左移，直至列表为空。那时，$#将变成 0，循环也将终止。

break 命令 内置命令 break 用于强行退出循环，但不退出程序(要退出程序，用 exit 命令)。执行 break 命令后，控制从关键字 done 的下一行开始。break 命令使控制从最内层循环退出来，因此，如果有嵌套循环，可以给 break 命令一个数字作为参数，这样就能指定跳到哪个外层循环外面。如果您嵌套了 3 层循环，最外层循环就是第 3 层循环，中间那层是第 2 层循环，最里面那层则是第 1 层循环。break 在退出无限循环时很有用。

格式

```
break [n]
```

范例 14-48

```
#!/bin/bash
  # Scriptname: loopbreak

1  while true; do
2      echo Are you ready to move on\?
       read answer
3      if [[  "$answer"  ==  [Yy] ]]
       then
4          break
5      else
           ...commands...
       if
6   done
7   print "Here we are"
```

说明

1. true 命令是一个 UNIX/Linux 命令，它总是以状态 0 退出。true 命令常常被用来启动无限循环。如果有分号分隔，就可以把 do 语句和 while 命令写在同一行上。由于 true 返回真，控制进入循环体。

2. 要求用户输入，把用户的输入赋给变量 answer。

3. 如果变量 answer 的值是 Y 或 y，控制便转到第 4 行。

4. 执行 break 命令，退出循环，控制转到第 7 行。显示一行文本："Here we are"。除非用户回答 Y 或 y，否则程序将继续要求用户输入，循环将永远执行下去。

5. 如果第 3 行的测试失败，就执行 else 命令。当循环体在关键字 done 处终止时，控制再次从第一行的 while 顶部开始。

6. 这是循环体的结尾。

7. 执行 break 命令后，控制转到这里开始继续执行。

continue 命令 continue 命令在某个条件为真时，把控制转回循环的顶部。continue 下面的所有命令都被忽略。如果被嵌套在多层循环之内，continue 将控制转回最内层循环的顶部。如果给它一个数字作为参数，continue 就能将控制转到任一层循环的起点。例如



您嵌套了 3 层循环，最外层循环就是第 3 层循环，中间那层是第 2 层循环，最里面那层则是第 1 层循环[13]。

格式

```
continue [n]
```

范例 14-49

（邮件列表）

```
   $ cat mail_list
   ernie
   john
   richard
   melanie
   greg
   robin
```

（脚本）

```
   #!/bin/bash
   # Scriptname: mailem
   # Purpose: To send a list
1  for name in $(cat mail_list)
   do
2    if [[$name == richard]]; then
3        continue
     else
4          mail $name < memo
     fi
5  done
```

说明

1. 执行完命令替换$(cat mali_list)或`cat mail_list`后，for 循环开始遍历从文件 mail_list 中得到的名字列表。

2. 如果 name 的值等于 richard，就执行 continue 命令，控制转回循环顶部计算循环表达式的地方。richard 此时已被移出列表，所以赋给变量 name 的是下一个用户名，melanie。老式风格的写法是：`if [ "$name" = richard ] ; then`

3. continue 命令将控制转回循环顶部，跳过循环体中剩余的所有命令。

4. 给列表中除 richard 外的所有用户邮寄文件 memo 的一份副本。

5. 循环体的结尾。

嵌套循环和循环控制　使用嵌套循环时，可以给 break 和 continue 命令一个整型的数值参数，这样可以使控制从内层循环跳到外层循环。

---

⑬ 如果给 continue 命令的参数大于循环的层数，就退出整个循环。

范例 14-50

（脚本）

```
    #!/bin/bash
    # Scriptname: months
1   for month in Jan Feb Mar Apr May Jul Aug Sep Oct Nov Dec
    do
2     for week in 1 2 3 4
      do
          echo -n "Processing the month of $month.OK?"
          read ans
3         if ["$ans" = n -o -z "$ans"]
          then
4             continue 2
          else
              echo -n "Process week $week of $month?"
              read ans
              if ["$ans"=n -o -z "$ans"]
              then
5             continue
              else
              echo "Now processing week $week of $month."
              Sleep 1
                #Commands to here
              echo "Done processing..."
              fi
          fi
6     done
7   done
```

（输出）

```
Processing the month of Jan. OK?
Processing the month of Feb. OK? y
Process week 1 of Feb? y
Now processing  week 1 of Feb.
Done processing...
Processing the month of Feb. OK? y
Process week 2 of Feb? y
Now processing week 2 of Feb.
Done processing...
Processing the month of Feb. OK? n
Processing the month of Mar. OK? n
Processing the month of Apr. OK? n
Processing the month of May. OK? n
```

说明

1. 启动外层的 for 循环。执行第一轮循环时，把 Jan 赋给变量 month。

2. 启动内层的 for 循环。执行第一轮循环时，把 1 赋给变量 week。返回外层循环之前，内层循环将在对象列表上完整地遍历一遍。

3. 如果用户输入字母 n 或按下回车键，就执行第 4 行。



4. continue 命令的参数是 2，所以将控制转回从内往外数的第 2 层循环(本例中是最外层)。没有参数的 continue 变量则把控制转回最内层循环的顶部。

5. 控制返回最内层 for 循环的顶部。

6. 这个 done 终结最内层循环。

7. 这个 done 终结最外层循环。

14.6.7 I/O 重定向与子 shell

shell 可以通过管道或重定向将输入从文件输入改为从循环读取输入，也可以将输出到循环改为输出到文件。shell 启动一个子 shell 来处理 I/O 重定向和管道。循环结束后，在循环中定义的所有变量对脚本的其余部分都是不可见的。

将循环的输出重定向到一个文件　bash 循环的输出不仅可以送到屏幕，也可以通过管道送到文件中。参见范例 14-51。

范例 14-51

```
(命令行)
1   $ cat memo
    abc
    def
    ghi

(脚本)
    #!/bin/bash
    # Program name: numberit
    # Put line numbers on all lines of memo
2   if (( $# < 1 ))
    then
3      echo "Usage: $0 filename " >&2
       exit 1
    fi
4   count=1                      # Initialize count
5   cat $1 | while read line
    # Input is coming from file provided at command line
    do
6      ((count == 1)) && echo "Processing file $1..." > /dev/tty
7      echo -e "$count\t$line"
8      let count+=1
9   done > tmp$$                # Output is going to a temporary file
10  mv tmp$$ $1

(命令行)
11  $ numberit memo
    Processing file memo

12  $ cat memo
    1  abc
    2  def
    3  ghi
```

说明

1. 显示文件 memo 的内容。

2. 如果用户运行该脚本时没有提供命令行参数，参数个数($#)将小于 1，于是显示报错信息。

3. 如果参数个数小于 1，把解释命令用法的消息发到 stderr(>&2)。

4. 变量 count 被赋值为 1。

5. UNIX/Linux 的 cat 命令显示由$1 指定名称的文件的内容，cat 命令的输出通过管道传给 while 循环。执行第一轮循环时，赋给 read 命令的是文件的第一行，执行下一轮循环时则赋给它文件的第二行，以此类推。如果读输入成功，read 命令返回的退出状态是 0，失败时退出状态是 1。

6. 如果 count 的值是 1，就执行 echo 命令，把它的输出发往/dev/tty，即屏幕。

7. echo 命令显示 count 的值，后跟文件中该行的内容。

8. count 的值加 1。

9. 整个循环的输出，即$1 所指定的文件中的每一行，被重定向到文件 tmp$$，文件的第一行是个例外，它被重定向到终端，即/dev/tty[14]。

10. 文件 tmp$$被更名为$1 中保存的名称。

11. 执行该程序。被处理的文件名称为 memo。

12. 执行完脚本后，显示文件 memo 的内容，可以看到每一行前面都加上了它的行号。

把循环的输出通过管道传给 UNIX 命令 循环的输出可以通过管道传给另一条(组)命令，也可以重定向到一个文件。

范例 14-52

(脚本)
```
   #!/bin/bash
1  for i in 7 9 2 3 4  5
2  do
       echo  $i
3  done | sort -n
```

(输出)
```
2
3
4
5
7
9
```

说明

1. for 循环遍历一组未排序的整数。

2. 在循环体中，脚本输出这组整数。输出将通过管道传到 UNIX/Linux 的 sort 命令，按数值进行排序。

---

⑭ $$展开后是当前 shell 的 PID 号。把这个数字添在文件名的后面，可以使文件名唯一。



3. 在关键字 done 之后创建管道。循环将在子 shell 中执行。

14.6.8 在后台执行循环

循环可以在后台执行，这样可以继续执行其他程序而不必等待循环处理全部完成。

范例 14-53

```
(脚本)
   #!/bin/bash
1  for person in bob jim joe sam
   do
2      mail $person < memo
3  done &
```

说明

1. for 循环要移走词表中的所有名字，包括：bob、jim、joe 和 sam。它将每个名字依次赋给变量 person。
2. 在循环体中，向每个人发送文件 memo 的内容。
3. 关键字 done 后的与号使得循环在后台运行。当循环执行时，程序也继续往下执行。

14.6.9 IFS 和循环

shell 的内部字段分隔符(IFS)的值包括空格、制表符和换行符。IFS 被那些需要分析词列表的命令(如 read、set 和 for)用作词(标记)分隔符。如果列表使用其他不同的分隔符，用户也可以重新设置 IFS。改变 IFS 的值之前，最好先把它原来的值保存到另一个变量中。这样做的好处是，一旦需要，可以很方便的把 IFS 恢复为默认值。

范例 14-54

```
(The Script )
   #/bin/bash
   # Scriptname: runit2
   # IFS is the internal field separator and defaults to
   # spaces, tabs, and newlines.
   # In this script it is changed to a colon.

1  names=Tom:Dick:Harry:John
2  oldifs="$IFS"          # Save the original value of IFS

3  IFS=":"

4  for persons in $names
   do
5      echo  Hi $persons
   done

6  IFS="$oldifs"           # Reset the IFS to old value

7  set Jill Jane Jolene    # Set positional parameters
```

```
8   for girl in $*
    do
9       echo Howdy $girl
    done
```

(输出)

```
5   Hi Tom
    Hi Dick
    Hi Harry
    Hi John
9   Howdy Jill
    Howdy Jane
    Howdy Jolene
```

说明

1. 变量 names 被设置为字符串"Tom:Dick:Harry:John"，各个词之间用冒号分隔。

2. 把 IFS 的值(空白符)赋给另一个变量 oldifs。因为 IFS 的值是空白符，所以必须对它加引号进行保护。

3. 将 IFS 设置为冒号。现在，冒号被用来分隔词。

4. 变量替换完成后，for 循环以冒号作为词之间的内部字段分隔符，对每个名字进行遍历。

5. 显示词表中的所有名字。

6. 把 IFS 重新设成保存在 oldifs 中的原值。

7. 设置位置参量，把$1 赋值为 Jill，把$2 赋值为 Jane，$3 则赋值为 Jolene。

8. $*的值代表所有位置参量，即：Jill、Jane 和 Jolene。for 循环在每次迭代时，逐一将这些名字赋给变量 girl。

9. 显示参数列表中的每一个名字。

14.7 函数

函数是从 AT&T 的 UNIX System VR2 开始被引入 Bourne shell 的，bash shell 对函数又进行了加强。函数其实就是为某一条或某一组命令命名。函数用来模块化程序，并且使程序更有效率。函数在当前 shell 的环境中执行，也就是说，函数执行可执行程序如 ls 时并不派生子进程。您甚至可以把函数保存在另外的文件中，使用时再把它们载入您的脚本。

接下来让我们回顾一些使用函数的重要规则。

(1) 由 bash shell 来判断用的究竟是别名、函数、内置命令，还是磁盘上的可执行程序(或脚本)。它先在别名中找，然后是函数，内置命令，最后才是可执行程序。

(2) 函数必须先定义，后使用。

(3) 函数在当前环境中运行。函数共享调用它的脚本中的变量，还允许以给位置参量赋值的方式向函数传递参数。可以使用 local 功能在函数内部创建局部变量。

(4) 如果在函数中使用 exit 命令，就会退出整个脚本。如果只是从函数中退出，就只



是返回到脚本调用函数的地方。

(5) 函数中的 return 语句，返回函数执行的最后一条命令的退出状态，或者返回指定的参数值。

(6) 使用内置命令 export -f 可以把函数导出到子 shell 中。

(7) 使用 declare -f 命令可以列出函数名及其定义，使用 declare -F 命令只是列出函数名[15]。

(8) 陷阱(trap)和变量一样，被不同的函数共用。它们被脚本和脚本调用的函数共享。如果在函数中定义了一个陷入，这个陷入也会被脚本共享。这种机制可能会导致一些令人讨厌的副作用。

(9) 如果函数保存在其他文件中，可以用 source 或 dot 命令把它们载入到当前脚本中。

(10) 函数可以递归，即调用它们本身。且递归调用的次数没有限制。

格式

```
function 函数名() { 命令; 命令; }
```

范例 14-55

```
function dir { echo "Directories: ";ls -l|awk '/^d/ {print $NF}'; }
```

说明

关键字 function 后紧跟函数名 dir(有时函数名后跟有空圆括号，但不是必须的)。键入 dir 时，shell 将执行花括号里的命令。这个函数的功能是只列出当前工作目录下的子目录。花括号两侧的空格是必需的。

14.7.1 清除函数

从内存中清除某个函数，使用 unset 命令。

格式

```
  unset  函数名
```

14.7.2 导出函数

可以将函数导出，使它们在子 shell 可用。

格式

```
export -f  函数名
```

14.7.3 函数的参数和返回值

由于函数是在当前 shell 中执行的，所以变量对函数和 shell 都是可见的。在函数中对环境所做的任何改动也会对 shell 的环境生效。

参数 可以使用位置参量向函数传递参数。位置参量是函数私有的，也就是说，函数对参数的操作不会影响在函数外使用的任何位置参量。见范例 14-56。

内置 local 功能 函数私有的局部变量，在函数退出后随之消失，可以使用内置的 local

---

[15] 仅在 Bash 版本 2.x 中有效

功能创建局部变量。见范例 14-57。

内置的 return 命令　return 命令可用来退出函数并将控制转回程序调用函数的位置(注意，在脚本的任何位置使用 exit，包括在函数内，都会终止脚本的运行)。如果没有特别指定参数，函数的返回值其实就是函数中最后一条命令的退出状态。如果给 return 命令赋一个值，该值就被保存在变量?中，这个值可以是一个 0~255 之间的整数。因为规定了 return 命令只能返回 0~255 之间的整数，所以可以用命令替换来获取函数的输出。具体做法与获取 UNIX 命令的输出时所做的一样：把函数名放在由$符引导的括弧中(如：$(function_name))，或是用传统的方式把函数名放在一对反引号之间，这样就可以将结果赋给某个变量。

范例 14-56

(传递参数)

(脚本)

```
    #!/bin/bash
    # Scriptname: checker
    # Purpose: Demonstrate function and arguments

1   function Usage { echo "error: $*" 2>&1; exit 1; }

2   if (( $# != 2 ))
    then
3       Usage "$0: requires two arguments"
    fi
4   if [[ ! ( -r $1 && -w $1 ) ]]
    then
5       Usage "$1: not readable and writable"
    fi
6   echo The arguments are: $*
 <  Program continues here >
```

(命令行和输出)

```
$ checker
error: checker: requires two arguments
$ checker file1 file2
error: file1: not readable and writable
$ checker filex file2
The arguments are filex file2
```

说明

1. 定义 Usage 函数。该函数用来向标准错误输出(屏幕)发送一条错误信息。函数的参数由调用函数时的字符串组成，参数保存在变量$*中。$*是一个特殊的变量，保存着函数内的所有位置参量。在函数内部，位置参量都是局部变量，与函数外使用的位置参量没有关系。

2. 如果从命令行传递给脚本的参数个数不等于 2，程序转到第 3 行的分支上。

3. 调用 Usage 函数时，给出的字符串“$0: requires two arguments”传送给函数，并保存在 $* 变量中，echo 语句把该信息发送到标准错误输出上，且脚本以退出状态 1 退出，



表示出错了[16]。

4，5. 如果从命令行接受的第一个参数不是一个可读写文件的文件名，则调用Usage 函数，函数的参数是"$1: not readable and writeable"。

6. 脚本从命令行接受的参数保存在变量$*中，它们与函数内部的$*变量没 有关系。

范例 14-57

（使用 return 命令）

（脚本）

```
    #!/bin/bash
    # Scriptname: do_increment
1   increment () {
2     local sum;     # sum is known only in this function
3     let "sum=$1 + 1"
4     return $sum    # Return the value of sum to the script
    }
5   echo -n "The sum is "
6   increment 5   # Call function increment; pass 5 as a
                  # parameter; 5 becomes $1 for the increment function
7   echo $?       # The return value is stored in $?
8   echo $sum     # The variable "sum" is not known here
```

（输出）

```
4,6  The sum is 6
8
```

说明

1. 定义函数 increment。

2. 内置的 local 功能定义函数的局部私有变量 sum，sum 在函数外不可见，函数退出时，它也随之消失。

3. 调用函数时，第一个参数的值(即$1)被加 1，相加的结果被赋给变量 sum。

4. 如果指定了参数，内置命令 return 将返回主脚本中函数被调用时所处位置的下一行，同时将它的参数保存在变量？中。

5. 把这个字符串回显在屏幕上。

6. 用 5 作为参数调用 increment 函数。

7. 函数返回时，它的退出状态被保存在变量？中。如果 return 语句没有指定参数，函数返回的就是函数中最后一条命令的退出状态值。return 命令的参数必须是一个 0~255 之间的整数。

8. 变量 sum 是在函数 increment 中定义的，它的作用域却是局部的，在调用函数外部是不可见的，因此这一行显示空值。

范例 14-58

（使用命令替换）

（脚本）

---

⑯ 这一行的语句如果用老的 test 命令的形式，表达式应写为： if [! \(-r $1 -a -w $1)] 。

```
    #!/bin/bash
    # Scriptname: do_square
1   function square {
      local sq    # sq is local to the function
      let "sq=$1 * $1"
      echo "Number to be squared is $1."
2       echo "The result is $sq "
    }
3   echo "Give me a number to square. "
    read number
4   value_returned=$(square $number)  # Command substitution
5   echo   "$value_returned"
```

(命令行与输出)

```
    $ do_square
3   Give me a number to square.
    10
5   Number to be squared is 10.
    The result is 100
```

说明

1. 定义函数 square。被调用时，它的功能是计算出参数(即$1)的平方(即参数自乘)。

2. 显示将该数平方(自乘)后所得的结果。

3. 要求用户输入一个整数。程序从这一行开始执行。

4. 用(用户输入的)一个数作为参数来调用函数 square。因为函数被括在由$符引导的括号之间，所以 shell 执行命令替换。函数的输出，即两条 echo 语句的输出，被赋值给变量 value_returned。

5. 显示命令替换后返回的值。

14.7.4 函数与 source(或 dot)命令

保存函数 函数常常被定义在.profile 文件中，这样，用户登入系统时，函数就被自动定义。函数能被导出，也可以被保存在文件中。所以，当需要使用某个函数时，只要指定文件名作为参数，调用 source 或 dot 命令激活该文件中的函数定义就可以了。

范例 14-59

```
1   $ cat myfunctions
2   function go() {
        cd $HOME/bin/prog
        PS1='`pwd` > '
        ls
    }
3   function greetings() { echo "Hi $1! Welcome to my world." ; }
4   $ source myfunctions
5   $ greetings george
    Hi george! Welcome to my world.
```

说明

1. 显示文件 myfunctions 的内容，其中包含两个函数定义。



2. 定义的第一个函数名为 go，它的功能是将主提示符设置为当前工作目录。

3. 定义的第二个函数名为 greetings，它将问候由参数指定的用户。

4. source 或 dot 命令把文件 myfunctions 的内容加载到 shell 的内存空间。现在两个函数都在当前 shell 中定义了。

5. 调用并执行 greetings 函数。

范例 14-60

```
(The dbfunctions file shown below contains functions to be used by the main
program. See cd for complete script.)

1   $ cat dbfunctions
2   function addon () {     # Function defined in file dbfunctions
3       while true
        do
            echo "Adding information "
            echo "Type the full name of employee "
            read name
            echo "Type address for employee "
            read address
            echo "Type start date for employee (4/10/88 ) :"
            read startdate
            echo $name:$address:$startdate
            echo -n "Is this correct? "
            read ans
            case "$ans" in
            [Yy]*)
                echo "Adding info..."
                echo $name:$address:$startdate>>datafile
                sort -u datafile -o datafile
                echo -n "Do you want to go back to the main menu? "
                read ans
                if [[ $ans == [Yy] ]]
                then

4               return     # Return to calling program
                else
5               continue   # Go to the top of the loop
                fi
                ;;
            *)
            echo "Do you want to try again? "
            read answer
            case "$answer" in
            [Yy]*) continue;;
            *) exit;;
            esac
                ;;
        esac
    done
```

```
6   }     # End of function definition

--------------------------------------------------------------------
(命令行)
7   $ more mainprog
    #!/bin/bash
    # Scriptname: mainprog
    # This is the main script that will call the function, addon

    datafile=$HOME/bourne/datafile
8   source dbfunctions     # The file is loaded into memory
       if [ ! -e $datafile ]
    then
       echo "$(basename $datafile) does not exist" >&2
       exit 1
    fi
9      echo "Select one: "
       cat <<EOF
          [1] Add info
          [2] Delete info
          [3] Update info
          [4] Exit
       EOF
       read choice
       case $choice in
10        1)   addon     # Calling the addon function
               ;;
          2)   delete    # Calling the delete function
               ;;
          3)   update
               ;;
          4)
               echo Bye
               exit 0
               ;;
          *)   echo Bad choice
               exit 2
               ;;
       esac
       echo Returned from function call
       echo The name is $name
       # Variable set in the function are known in this shell.
    done
```

说明

1. 显示文件 dbfunctions 的内容。

2. 定义函数 addon。它的功能是往文件 datafile 中添加新的信息。

3. 进入 while 循环。如果循环体中没有像 break 或 continue 这样的循环控制语句，它就会永远循环下去。



4. return 命令把控制转回程序中调用函数的位置。

5. 控制回到 while 循环的顶部。

6. 右花括号结束函数定义。

7. 这是主脚本。脚本中会用到函数 addon。

8. source 命令将文件 dbfunctions 了载入程序的内存空间。现在主脚本中已经有了函数 addon 的定义，可以使用它了。这样做的效果与直接在主脚本中定义函数一样。

9. 用 here 文档显示一个菜单，要求用户选择一个菜单项。

10. 调用 addon 函数。

14.8 捕获信号

如果在程序运行时按下 Ctrl+C 或 Ctrl+\组合键，程序就会在信号到达那一刻立即终止。有时候您可能不想让程序在信号到达后立即终止，您可以让程序忽略信号，继续运行或者先执行某些清理操作再退出脚本。trap 命令使您能够控制程序收到信号后的行为。

信号被定义为由一个整数构成的异步消息，它可以由某个进程发给另一个进程，也可以在用户按下某些特定的键或发生某种异常事件时，由操作系统发给某个进程[17]。trap 命令可以通知 shell：如果收到某个信号，就终止正在执行的命令。如果 trap 命令后面跟着括在引号中的命令，则收到指定信号时，shell 会执行这个命令串。shell 需要把这个命令串读两遍，设置信号陷阱时读一遍，信号到达时再读一遍。如果命令串两端是双引号，shell 会在第一次设置该陷阱时执行该命令串中所有的变量和命令替换。如果命令串两端是单引号，则其中的变量和命令替换要等到探测到信号后执行捕获时才会发生。

使用命令 kill –l 或 trap –l，将得到一个所有信号的列表。表 14-6 提供了信号编号与相应的信号名。

表 14-6 信号及其编号[18]

1) SIGHUP	9) SIGKILL	17) SIGCHLD	25) SIGXFSZ
2) SIGINT	10) SIGUSR1	18) SIGCONT	26) SIGVTALRM
3) SIGQUIT	11) SIGSEGV	19) SIGSTOP	27) SIGPROF
4) SIGILL	12) SIGUSR2	20) SIGTSTP	28) SIGWINCH
5) SIGTRAP	13) SIGPIPE	21) SIGTTIN	29) SIGIO
6) SIGABRT	14) SIGALRM	22) SIGTTOU	30) SIGPWR
7) SIGBUS	15) SIGTERM	23) SIGURG	
8) SIGFPE	16) SIGSTKFLT	24) SIGXCPU	

⑰ 参见 Morris I.Bolsky 与 David G. Korn 所著的 *The New KornShell Command and Programming Language*(Englewood Cliffs, NJ: Prentice Hall PTR, 1995)一书的第 327 页。

⑱ 有关 UNIX 信号及其含义的完整列表，请访问 www.cybermagician.com.uk/technet/unixsignals.htm。Linux 信号请访问 www.comptechdoc.org/os/linux/programming/linux_pgsignals.html。

格式

```
trap '命令; 命令' 信号编号
trap '命令; 命令' 信号名
```

范例 14-61

```
trap 'rm tmp*; exit 1'  0 1 2 15
trap 'rm tmp*; exit 1'  EXIT HUP INT TERM
```

说明

若信号 1(挂起)、信号 2(中断)和信号 15(软件终止)中的任何一个到达，则删除所有临时文件，然后退出。

如果脚本在运行过程中收到中断信号，可以使用 trap 命令来选择不同的处理方式：正常处理该信号(默认方式)、忽略该信号或创建一个处理函数以在信号到达时调用。

像 HUP 和 INT 这样的信号名，通常带有前缀 SIG，如 SIGHUP，SIGINT 等[19]。Bash shell 允许使用不带 SIG 前缀的信号名，或是直接使用信号的编号。参见表 14-6，伪信号名 EXIT，或数字 0 都会引起执行捕获信号而导致 shell 退出。

14.8.1 重置信号

要将信号重置为按默认方式处理，在 trap 命令后面跟上信号的名称或编号即可。在函数中设置的陷阱，只要函数一运行，它们对调用函数的 shell 就是可见的，而所有在函数外设置的陷阱，对函数也是可见的。

范例 14-62

```
trap INT
```

说明

重置信号 2(即 SIGINT)的默认动作。该信号的默认动作是在按下中断键(Ctrl+C 组合键)时终止进程。

范例 14-63

```
trap 2
```

说明

信号 2(SIGINT)的默认动作被复位，该信号是用来终止进程的(例如使用 Ctrl+C 组合键)

范例 14-64

```
trap 'trap 2' 2
```

说明

设置信号 2(SIGINT)的默认动作，以在信号到达时执行引号中的命令。用户按 Ctrl+C 组合键两次将终止程序。第一次 trap 捕获到信号，第二次 trap 将陷阱复位成默认动作，默认为终止进程。

---

⑲ SIGKILL 的值为 9，通常称为“终止”信号，该信号是不能被捕获的。



14.8.2 忽略信号

如果 trap 命令后面跟的是一对空引号，则其后所列的信号都将被进程忽略。

范例 14-65

```
trap " " 1 2   或   trap "" HUP INT
```

说明

信号 1(SIGHUP)和信号 2(SIGINT)将被 shell 忽略。

14.8.3 列出陷阱

键入 trap 命令，就能列出所有陷入和为它们指定的命令。

范例 14-66

(在命令行上)

```
1  $ trap 'echo "Caught ya!; exit"' 2
2  $ trap
   trap -- 'echo "Caught ya!; exit 1"' SIGINT
3  $ trap -
```

说明

1. 对信号 2 设置陷阱命令，在捕获到信号 2(按下 Ctrl+C 组合键)时退出。
2. 不带参数的 trap 命令列出所有已设置的陷阱。
3. 如果 trap 命令的参数是一个短划线，则重置所有信号为其初始值，也就是信号重置为 shell 启动时的值。

范例 14-67

(脚本)

```
   #!/bin/bash
   # Scriptname: trapping
   # Script to illustrate the trap command and signals
   # Can use the signal numbers or bash abbreviations seen
   # below. Cannot use SIGINT, SIGQUIT, etc.
1  trap 'echo "Ctrl-C will not terminate $0."'  INT
2  trap 'echo "Ctrl-\\ will not terminate $0."'  QUIT
3  trap 'echo "Ctrl-Z will not terminate $0."'  TSTP
4  echo  "Enter any string after the prompt.
   When you are ready to exit, type \"stop\"."
5  while true
   do
6     echo -n "Go ahead...> "
7     read
8     if [[ $REPLY == [Ss]top ]]
      then
9        break
      fi
10  done
```

（输出）
```
4  Enter any string after the prompt.
   When you are ready to exit, type "stop".
6  Go ahead...> this is it^C
1  Ctrl-C will not terminate trapping.
6  Go ahead...> this is it again^Z
3  Ctrl-Z will not terminate trapping.
6  Go ahead...> this is never it|^\
2  Ctrl-\\ will not terminate trapping.
6  Go ahead...> stop
   $
```

说明

1. 第 1 个 trap 命令捕获 INT 信号，即 Ctrl+C。如果用户在程序运行过程中按下^C，程序将执行引号中的命令。程序不会异常终止，而是显示“Ctrl-C will not terminate trapping”，并且继续提示用户输入。

2. 第 2 个 trap 命令将在用户按下 Ctrl+\组合键(发出 QUIT 信号)时被执行。程序将显示字符串“Ctrl–\\ will not terminate trapping”，然后继续运行。SIGQUIT 信号的默认动作是终止进程并生成 core 文件。

3. 第 3 个 trap 命令将在用户按下 Ctrl+Z 组合键(发出 TSTP 信号)时被执行。程序将显示字符串“Ctrl–Z will not terminate trapping”，然后继续运行。这个信号的默认动作是：如果实现作业控制的话，该程序将在后台挂起。

4. 提示用户输入。

5. 进入 while 循环。

6. 显示提示“Go ahead…>”，程序等待输入(见下一行的 read 命令)。

7. read 命令将用户输入的内容赋给内置变量 REPLY。

8. 如果 REPLY 的值等于 Stop 或 stop，break 命令将导致循环退出并终止程序。除非用 kill 命令终止进程，否则，我们只能用输入 Stop 或 stop 的方法退出该程序。

9. break 命令使控制从循环体中退出。

10. done 标志着循环的结束。

14.8.4 函数中的信号陷阱

如果在函数中使用陷阱来处理某个信号，一旦该函数被调用，就会影响到整个脚本对该信号的处理，因为这个陷阱对脚本而言是全局的。下面这个范例中，陷阱被设置为忽略中断键^C。只有用 kill 命令杀死该脚本才能中断其中的循环。这个范例说明在函数中使用陷阱可能会产生不希望见到的副作用。

范例 14-68

（脚本）
```
   #!/bin/bash
1  function trapper () {
       echo "In trapper"
```




```
2       trap 'echo "Caught in a trap!"' INT
        # Once set, this trap affects the entire script. Anytime
        # ^C is entered, the script will ignore it.
    }
3   while :
    do
        echo "In the main script"
4       trapper
5       echo "Still in main"
6       echo "The pid is $$"
        sleep 5
7   done
```

(输出)

```
In the main script
In trapper
Still in main
The pid is 4267
Caught in a trap!       # User presses ^C
In the main script
In trapper
Still in main
The pid is 4267
Caught in a trap!       # User just pressed Ctrl-C
In the main script
```

说明

1. 定义 trapper 函数，函数中的所有变量和陷入对脚本是全局的。

2. trap 命令将忽略信号 2，即 INT(终端键)。如果按下 Ctrl+C 组合键，将显示"Caught in a trap!"消息，脚本继续永远执行下去。该脚本可以使用 kill 命令或 Ctrl+\组合键来终止。

3. 主脚本开始无限循环。

4. 调用函数 trapper。

5. 函数返回时，从此处开始执行。

6. PID 将显示出来。稍后可以使用这个号码在另一个终端上终止这个进程。例如，如果 PID 为 4267，使用 kill 4267 可以终止进程。

7. 程序暂停(睡眠)5 秒。

14.9 调试

带-n 选项的 bash 命令，能够对脚本进行语法检查，而不去实际运行其中的任何一条命令。如果脚本中存在语法错误，shell 就会报错。如果没有错误，shell 就不显示任何内容。

最常用的脚本调试手段是用带-x 选项的 set 命令，或是调用带-x 选项和脚本名为参数的 bash。调试选项列表见表 14-7。这些选项能使您跟踪脚本的执行。此时，shell 对脚本中每条命令的处理过程是：先执行替换，接着显示，然后再执行它。shell 显示脚本中的行时，

会在行首添上一个加号(+)。

表 14-7 调试选项

命 令	选 项	功 能
bash –x 脚本名	回显	在变量替换之后、执行命令之前，显示脚本的每一行
bash –v 脚本名	详细	在执行之前，按输入的原样打印脚本中各行
bash –n 脚本名	不执行	解释但不执行命令
set –x	打开回显	跟踪脚本的执行
set +x	关闭回显	关闭跟踪功能

如果打开 verbose 选项，或者在启动 shell 时加上-v 选项(bash –v 脚本名)，脚本中的每一行都会按它在脚本的样子原封不动地显示在屏幕上，然后被执行(参见第 15 章"调试 shell 脚本" 中的内容)。

范例 14-69

(脚本)

```
    #!/bin/bash
    # Scriptname: todebug

1   name="Joe Shmoe"
    if [[ $name == "Joe Blow" ]]
    then
        printf "Hello $name\n"
    fi

    declare -i num=1
    while (( num < 5 ))
    do
       let num+=1
    done
    printf "The total is %d\n", $num
```

(输出)

```
2 bash -x todebug
+ name=Joe Shmoe
+ [[ Joe Shmoe == \J\o\e\ \B\l\o\w ]]
+ declare -i num=1
+ ((  num < 5  ))
+ let num+=1
+ ((  num < 5  ))
+ let num+=1
+ ((  num < 5  ))
+ let num+=1
+ ((  num < 5  ))
+ let num+=1
+ ((  num < 5  ))
+ printf 'The total is %d\n,' 5
The total is 5
```



说明

1. 该脚本名为 todebug。可以通过打开-x 开关来观察脚本的运行。循环的每一次遍历都显示在屏幕上，变量每次被设置和修改后，其值也被打印出来。

2. 带-x 选项启动 Bash shell。回显选项被打开，脚本的每一行都被显示在屏幕上，行首增加了一个加号(+)。变量替换在显示之前执行。在显示的命令行的后面，打印命令的执行结果。

14.10　命令行

14.10.1　用 getopts 处理命令行选项

如果编写一个需要很多命令行选项的脚本，位置参量未必总是最有效的方式。例如，UNIX 的 ls 命令就可以使用很多命令行选项和参数(变量前面需要有一个短划线来引导。参数则不需要)。选项可以通过多种方式传递给程序，如：ls –laFi，ls –i –a –l –F，ls –ia –F 等。如果您的脚本需要参数，您可以为每个参数单独分配一个位置参量，如：ls –l –i –F，3 个以短划线引导的选项将被分别赋给$1、$2 和$3。但是，当用户用一个短划线选项列出所有的选项时，如 ls –liF，会出现什么情况呢？这时，-liF 将被整个赋给脚本中的$1。getopts 函数使我们能够用与 ls 程序相同的方法来处理选项和参数[30]。getopts 函数使 runit 程序能够处理各种方式组合的参数。

范例 14-70

```
(The Command Line )
1  $ runit -x -n  200 filex
2  $ runit -xn200 filex
3  $ runit -xy
4  $ runit -yx -n 30
5  $ runit -n250 -xy filey
(这些参数可以进行任何其他的组合)
```

说明

1. 程序 runit 用了 4 个参数：x 是选项；n 也是选项，但它后面需要跟一个数字作为参数；filex 则是一个独立的参量。

2. 程序 runit 将选项 x 和 n，以及数字参数 200 组合在一起。filex 也是一个参数。

3. 用 x 和 y 的选项组合调用程序 runit。

4. 调用程序 runit 时，选项 x 和 y 被组合在一起。选项 n 和数字参数 30 则分别被单独传递。

5. 调用程序 runit 时，将选项 n 和数字参数组合，选项 x 和 y 组合，filex 则单独传递。

在讨论程序 runit 的所有细节之前，我们检查程序中调用 getopts 的语句，看看 geropts 是如何处理参数的。

---

[30] 请参见 UNIX/linux 手册页(第 3 节)中关于 C 库函数 getopt 的内容。

范例 14-71

(runit 脚本中的一行)

```
while getopts :xyn: name
```

说明

x、y 和 n 是选项。在本例中，第一个选项由一个冒号引导，表示 getopts 不报告错误信息。如果选项后有一个冒号，表示该选项需要一个用空白符分开的参数。参数是一个不用短划线引导的词。-n 选项需要一个参数。

在命令行键入选项时前面都要加一个短划线。

遇到不含短划线的选项时，getopts 就认为选项列表已结束。

每次被调用时，getopts 都将找到的下一个选项放到变量 name 中(这个变量名是任意的)。如果遇到非法变量，getopts 就将 name 的值设为问号。

包含 getopts 函数的脚本　下面这些范例将说明 getopts 如何处理参数。

范例 14-72

(脚本)

```
   #!/bin/bash
   # Program: opts1
   # Using getopts -- First try --

1  while getopts xy options
   do
2     case $options in
3     x) echo "you entered -x as an option";;
      y) echo "you entered -y as an option";;
      esac
   done
```

(命令行)

```
4  $ opts1 -x
   you entered -x as an option
5  $ opts1 -xy
   you entered -x as an option
   you entered -y as an option
6  $ opts1 -y
   you entered -y as an option
7  $ opts1 -b
   opts1:  illegal option --  b
8  $ opts1 b
```

说明

1. getopts 命令被用作 while 命令的条件。getopts 命令后面列出了该程序的有效选项，即 x 和 y。循环体逐一测试每个选项。getopts 将每个选项去掉短划线后赋给变量 options。所有的参数都处理完之后，getopts 将以一个非零的状态退出，从而终止 while 循环。

2. case 命令被用来测试在变量 options 中找到的每个可能选项，即 x 或 y。

3. 如果某个 x 是选项，则显示字符串“You entered x as an option”。



4. 在命令行给脚本 opts1 一个 x 选项，x 是一个合法选项，将被 getopts 处理。

5. 在命令行给脚本 opts1 一个 xy 选项，x 和 y 都是合法选项，都将被 getopts 处理。

6. 在命令行给脚本 opts1 一个 y 选项，y 是一个合法选项，将被 getopts 处理。

7. 给脚本 opts1 一个 b 选项，这是一个非法选项。getopts 将向 stderr 发送一条报错信息。

8. 前面没有短划线引导的选项不是 getopts 要处理的选项，于是 getopts 停止处理参数。

范例 14-73

(脚本)

```
    #!/bin/bash
    # Program: opts2
    # Using getopts -- Second try --

1   while getopts xy options 2> /dev/null
    do
2       case $options in
        x) echo "you entered -x as an option";;
        y) echo "you entered -y as an option";;
3       \?) echo "Only -x and -y are valid options"  1>&2;;
        esac
    done
```

(命令行)

```
    $ opts2 -x
    you entered -x as an option
    $ opts2 -y
    you entered -y as an option
    $ opts2 xy
    $ opts2 -xy
    you entered -x as an option
    you entered -y as an option
4   $ opts2 -g
    Only -x and -y are valid options
5   $ opts2 -c
    Only -x and -y are valid options
```

说明

1. 如果 getopts 发出报错信息，将该信息重定向到/dev/null。

2. 遇到非法选项时，getopts 将变量 options 赋值为一个问号。case 命令被用来测试问号，使您能够在标准错误输出上打印出自定义的报错信息。

3. 如果变量 options 被赋值为问号，就执行这条 case 语句。反斜杠用来保护问号，这样 shell 就不会把问号当成自己的通配符而对它执行文件名替换。

4. g 不是合法选项。因此，变量 options 被赋值为问号，屏幕上显示出报错信息。

5. c 不是合法选项。因此，变量 options 被赋值为问号，屏幕上显示出报错信息。

特殊的 getopts 变量 getopts 函数提供两个变量来记录参数的信息：OPTIND 和 OPTARG。OPTIND 是一个专用变量，初始化为 1，每次 getopts 处理完一个命令行参数后，OPTIND 就增加为 getopts 要处理的下一个参数的序号。OPTARG 变量包含了合法参数的

值。参见范例 14-74 和 14-75。

范例 14-74

(脚本)

```
    #!/bin/bash
    # Program: opts3
    # Using getopts -- Third try --

1   while getopts dq: options
    do
        case $options in
2           d) echo "-d is a valid switch ";;
3           q) echo "The argument for -q is $OPTARG";;
            \?) echo "Usage:opts3 -dq filename ... " 1>&2;;
        esac
    done
```

(命令行)

```
4   $ opts3 -d
    -d is a valid switch
5   $ opts3 -q  foo
    The argument for -q is foo
6   $ opts3 -q
    Usage:opts3 -dq filename ...
7   $ opts3 -e
    Usage:opts3 -dq filename ...
8   $ opts3 e
```

说明

1. while 命令测试 getopts 的退出状态。如果 getopts 成功地处理了一个参数，它将返回退出状态 0，于是程序进入 while 循环的循环体。参数列表后面的冒号表示 q 选项必须带一个参数，参数将保存在特殊变量 OPTARG 中。

2. d 是一个合法选项。如果把 d 作为一个选项输入，不带短划线的 d 被保存在 options 变量中。

3. q 是一个合法选项。q 选项需要一个参数，在 q 选项及其参数之间必须有空白符。如果把 q 作为一个选项输入并带一个参数，不带短划线的 q 将保存到 options 变量中，而参数则保存到 OPTARG 变量中。如果 q 选项后没有跟参数，则把一个问号保存到变量 options 中。

4. d 选项是 opts3 的一个合法选项。

5. 带参数的 q 选项也是 opts3 的一个合法选项。

6. 不带参数的 q 选项是错误的。

7. e 选项是无效选项。如果选项是非法的，则把一个问号保存到变量 options 中。

8. 选项既不带短划线作引导符，也不带加号作引导符，这时，getopts 将不把它作为选项处理，而返回一个非 0 退出状态。While 循环结束。



范例 14-75

(脚本)

```
    #!/bin/bash
    # Program: opts4
    # Using getopts -- Fourth try --

1   while getopts xyz: arguments 2>/dev/null
    do
        case $arguments  in
2        x) echo "you entered -x as an option .";;
         y) echo "you entered -y as an option." ;;
3        z) echo "you entered -z as an option."
            echo "\$OPTARG is $OPTARG.";;
4        \?) echo "Usage opts4 [-xy] [-z  argument]"
            exit 1;;
        esac
    done
5   echo " The number of arguments passed was $(( $OPTIND - 1 ))"
```

(命令行)

```
$ opts4 -xyz foo
You entered -x as an option.
You entered -y as an option.
You entered -z as an option.
$ OPTARG is foo.
The number of arguments passed was 2.

$ opts4 -x -y -z  boo
You entered -x as an option.
You entered -y as an option.
You entered -z as an option.
$OPTARG is boo.
The number of arguments passed was 4.

$ opts4 -d
Usage: opts4  [-xy] [-z argument]
```

说明

1. while 命令测试 getopts 的退出状态。如果 getopts 成功地处理了一个参数，它将返回退出状态 0，于是程序进入 while 循环的循环体。z 选项后面的冒号表示-z 选项必须带一个参数，参数将保存在 getopts 的内置变量 OPTARG 中。

2. 如果 x 作为选项被传入脚本，将被保存在变量 arguments 中。

3. 如果z作为带参数的选项被传入脚本，则它的参数将被保存在内置变量OPTARG中。

4. 如果传入的是无效选项，变量 arguments 中保存的将是一个问号，脚本显示一条报错信息。

5. getopts 的专用变量 OPTIND 保存的是待处理的下一个选项的编号。它的值总是比实际的命令行参数个数多 1。

14.10.2 eval 命令和命令行解析

eval 命令处理命令行，先执行所有的 shell 替换，然后执行命令行。当通常的命令行解析无法满足要求时，就要使用 eval 命令。

范例 14-76

```
1  $ set a b c d
2  $ echo The last argument is \$$#
3  The last argument is $4

4  $ eval echo The last argument is \$$#
   The last argument is d

5  $ set -x
   $ eval echo The last argument is \$$#
   + eval echo the last argument is '$4'
   ++ echo the last argument is d
   The last argument is d
```

说明

1. 设置 4 个位置参量。

2. 用户希望得到的结果是显示最后一个位置参量的值。\$打印出一个美元符。$#的值是 4，即位置参量的个数。shell 求出$#的值后，不会再次解析命令行以得出$4 的值。

3. 显示的结果是$4，而不是最后那个参数。

4. shell 执行完所有的变量替换后，eval 命令又执行一次变量替换，然后执行 echo 命令。

5. 打开反显选项来观察解析的顺序。

范例 14-77

(来自 Shutdown 程序)

```
1  eval `/usr/bin/id | /usr/bin/sed 's/[^a-z0-9=].*//'`
2  if [ "${uid:=0}" -ne 0 ]
   then
3      echo $0: Only root can run $0
       exit 2
   fi
```

说明

1. 这个范例难度较高。id 程序的输出送给 sed，以提取出字符串中 uid 部分。id 的输出结果是：

```
id=9496(ellie) gid=40 groups=40
uid=0(root) gid=1(daemon) groups=1(daemon)
```

sed 命令中正则表达式的含义是：从字符串首开始，找出任何一个不是字母、数字或等号的字符，删除该字符和它后面的所有字符。结果是将第一个左圆括号到行尾的所有内容替换为空。这样处理后所剩的内容是 uid=9496 和 uid=0。

eval 完成对命令行的转换后，将执行所得的命令：uid=9496 或 uid=0。



例如，如果用户的 ID 是 root，eval 执行的命令将是 uid=0。这条命令将在脚本中创建一个名为 uid 的局部变量，并将它赋值为 0。

2. 测试变量 uid 的值是不为 0，则使用命令修饰符。
3. 如果 uid 的值不为 0，则 echo 命令显示脚本名($0)和这条消息。

14.11 bash 的选项

14.11.1 shell 调用选项

用 bash 命令启动 shell 时，可以通过选项来改变它的行为。有两种类型的选项：单字符选项和多字符选项。单字符选项由一个短划线和单个字符组成，多字符选项由两个短划线和任意个字符组成。同时使用时，多字符选项必须放在单字符选项前。一个交互式登录 shell 通常用下列选项启动：-i(启动一个交互式 shell)、-s(从标准输入读)和-m(启动作业控制)。见表 14-8。

表 14-8 2.x 版 bash shell 的调用选项

选　项	含　义
-c string	从字符串中读取命令。string 后的所有参数都被赋给从$0 开始的位置参量
-D	在标准输出上显示一个由$符引导的双引号括着的字符串列表。当当前环境不是 C 或 POSIX 时，这些字符串输出为一个翻译文本。同时隐含带有-n 选项，不执行任何命令
-i	shell 在交互模式下运行。忽略 TERM、QUIT 和 INTERRUPT 信号
-s	从标准输入读取命令，并允许设置位置参量
-r	启动一个受限 shell
--	选项结束的标志，不再处理后面的选项，而把--或-后面的内容当作文件名或参数
--dump-strings	同-D
--help	显示内置命令的使用信息，然后退出
--login	把 bash 作为一个登录 shell 调用
--noediting	当 bash 以交互方式运行时，不使用 Readline 库
--noprofile	启动后，bash 不读入下列初始化文件：/etc/profile，~/.bash_profile，~/.bash_login 和~/.profile
--norc	对交互式 shell，bash 不读入~/.bashrc 文件。如果用 sh 命令运行 shell 时，该选项默认时是打开的
--posix	当 bash 的行为不符合 POSIX 1003.2 标准时，就改变它使之符合该标准
--quiet	在 shell 启动时不显示任何信息，该选项默认情况下打开
--rcfile file	如果 bash 以交互式运行，使用指定的初始化文件而不是~/.bashrc 文件
--restricted	启动一个受限的 shell
--verbose	打开显示详细信息选项，同-v
--version	显示本 bash shell 的版本信息，然后退出

表 14-9　2.x 以前版本的 bash shell 的调用选项

选　项	含　义
-c string	从字符串中读取命令。string 后的所有参数都被赋给从$0 开始的位置参量
-D	在标准输出上打印一个由$符引导的双引号括着的字符串列表。当当前环境不是 C 或 POSIX 时，这些字符串输出为一个说明文本。同时隐含带有-n 选项，不执行任何命令
-i	shell 在交互模式下运行。忽略 TERM、QUIT 和 INTERRUPT 信号
-s	从标准输入读取命令，并允许设置位置参量
-r	启动一个受限 shell
-	选项结束的标志，不再处理后面的选项，而把--或-后面的内容当作文件名或参数。
-login	把 bash 作为一个登录 shell 调用
-nobraceexpansion	取消花括号扩展
-nolineediting	当 bash 以交互方式运行时，不使用 Readline 库
-noprofile	启动后，bash 不读入下列初始化文件：/etc/profile，~/.bash_profile，~/.bash_login 和~/.profile
-posix	当 bash 的行为不符合 POSIX 1003.2 标准时，就改变它使之符合该标准
-quiet	在 shell 启动时不显示任何信息，该选项默认情况下打开
-rcfile file	如果 bash 以交互式运行，使用指定的初始化文件而不是~/.bashrc 文件
-verbose	打开显示详细信息选项，同-v
-version	显示本 bash shell 的版本信息，然后退出

14.11.2　set 命令及其选项

除了处理命令行参数外，set 命令还可用来打开和关闭 shell 的选项。要打开某个选项，就在选项前加个短划线(-)。要关闭某个选项，则在选项前加上加号(+)。请参见表 14-10 中列出的 set 选项。

范例 14-78

```
1   $ set -f
2   $ echo *
    *
3   $ echo ??
    ??
4   $ set +f
```

说明

1. 打开 f 选项，禁止文件名扩展。
2. 星号没有被展开。
3. 问号没有被展开。
4. 关闭 f 选项，允许文件名扩展。



表 14-10 set 命令的选项

选 项 名	开关	含 义
allexport	-a	自动标出选项设置以来的被导出的新变量或被修改的变量，直至选项被取消
braceexpand	-B	打开花括号扩展，是默认设置[21]
emacs		使用内置的 emacs 编辑器进行命令行编辑时，是默认设置
errexit	-e	如果命令返回的状态非 0(命令失败)，则退出程序。在读入初始化文件时，不设置此选项
histexpand	-H	在处理历史替换时，启用!和!!。是默认选项[21]
history		启用命令行历史功能。是默认选项[21]
ignoreeof		禁止用 EOF(Ctrl+D)退出 shell，而必须用 exit 命令来退出 shell。等价于设置 shell 变量：IGNOREOF=1
keyword	-k	把所有的关键字参数放到命令的环境中
interactive-comments		在交互式 shell 中，把一行中#符后面的文本作为注释
monitor	-m	允许作业控制
noclobber	-C	当使用重定向时，保护文件不被覆盖
noexec	-n	读命令，但不执行，用于检查脚本的语法。在交互式执行时，选项关
noglob	-d	禁止路径名扩展，即关闭通配符
notify	-b	当后台作业完成时提醒用户
nounset	-u	扩展一个未设置的变量时显示错误信息
onecmd	-t	读取并执行完命令就退出[21]
physical	-P	设置后，在键入 cd 或 pwd 命令时不使用符号链接，而用物理路径代替
posix		如果 shell 的默认操作不符合 POSIX 标准，就改变 shell 的行为。将 shell 的默认行为设置为符合 POSIX1003.2 标准
privileged	-p	设置后，shell 不读取.profile 或 ENV 文件，且不从环境继承函数。在 setuid 脚本中自动设置
verbose	-v	打开 verbose 模式，用于调试
vi		使用 vi 作为内置编辑器进行命令行编辑
xtrace	-x	打开回显模式，用于调试

14.11.3 shopt 命令及其选项

2.x 版 bash 的 shopt 命令也可以用来打开或关闭 shell 的选项。参见表 14-11。

㉑ 选项只适用于 bash 2.x。

表 14-11 shopt 命令选项

选 项	含 义
cdable_vars	如果内置命令 cd 的参数不是一个目录，则该参数被当作一个变量名，变量的值是将要切换到的目录名
cdspell	纠正 cd 命令中简单的目录名拼写错误。可以检查的错误包括颠倒顺序的字符，遗漏的字符以及多余的字母等。如果发现了一个错误，则打印纠错后的路径名，并执行命令。只是在交互式 shell 中使用
checkhash	在执行命令前，bash 先在 hash 表中查找命令是否存在。如果不存在，再进行路径搜索
checkwinsize	在每一个 echo 命令后，bash 检查窗口的大小，如果有必要的话，还要改变内置变量 LINES 和 COLUMNS 的值
cmdhist	bash 将在同一历史项中保存一个多行命令的所有行。这使得重新编辑一个多行命令更容易
dotglob	bash 在文件名展开的结果中包含以点号(.)开头的文件名
execfail	如果非交互式 shell 不能执行内置命令 exec 的参数指定的文件，则该 shell 不会退出。如果 exec 执行失败，则交互式 shell 也不会退出
expand_aliases	扩展命令别名。默认情况下打开
extglob	使用扩展的模式匹配特性(源自 Korn shell，使用正则表达式元字符进行文件名扩展)
histappend	在 shell 退出时，把命令历史清单追加到 HISTFILE 所指定的文件中，而不是覆盖该文件
histreedit	如果使用了 readline，用户还有机会重新编辑一个失败的历史命令替换
histverify	如果使用了 readline，历史命令替换的结果并不立即送到 shell 解析器，而是把结果行装入 readline 编辑缓冲区，以便作进一步的修改
hostcomplete	设置后，在使用 readline 的情况下，如果正在输入包含@符的词，则 bash 将自动完成@符后面的主机名的拼写。默认情况下打开
huponexit	设置后，如果交互式 shell 退出，bash 就向所有的作业发送 SIGHUP 信号
interactive_comments	在交互式 shell 中，使一行上以#符开头的所有词都被忽略。默认情况下打开
lithist	如果同 cmdhist 选项一起设置，那么每个多行命令保存到命令历史中时，在可能的情况下，将用嵌入的换行符代替分号分隔符
mailwarn	设置后，bash 在检查邮件时，如果所检查的文件是上次已经检查过的，则显示信息“The mail in mailfile has been read”
nocaseglob	设置后，bash 在执行文件名展开时，将不区分大小写的方式匹配文件名
nullglob	设置后，bash 的文件名模式如果没有文件与之匹配，则展开为一个空字符串，而不是展开为它本身
promptvars	设置后，变量和参数展开后，将打印出来。默认情况下打开
restricted_shell	Shell 以受限模式启动时，此选项将被设置。不能修改它的值。启动文件执行时，它不被重置，从而启动文件可以查看 shell 是否受限



(续表)

选　项	含　义
shift_verbose	设置后，如果内置命令 shift 移动的个数超过了位置参量的总个数，则 shift 命令打印一条出错信息
sourcepath	设置后，内置命令 source 使用 PATH 的值来寻找文件所在的目录，该文件由 sourec 命令的参数指定。默认情况下打开
source	与点(dot)同义

14.12 shell 的内置命令

shell 有很多内置在其源代码中的命令。这些命令是内置的，所以 shell 不必到磁盘上搜索它们，执行速度因此加快。bash 提供的 help 功能，能提供任何内置命令的在线帮助，表 14-12 列出了这些内置命令。

表 14-12　内置命令

命　令	功　能
.	执行当前进程环境中的程序。同 source
. file	dot 命令从文件 file 中读取命令并执行
:	空操作，返回退出状态 0
alias	显示和创建已有命令的别名
bg	把作业放到后台
bind	显示当前关键字与函数的绑定情况，或将关键字与 readline 函数或宏进行绑定
break	从最内层循环跳出
break [n]	请参见 14.6 节“break 命令”
builtin [sh-builtin [args]]	运行一个内置 shell 命令，并传送参数，返回退出状态 0。如果一个函数与一个内置命令同名时，该命令将很有用[20]
cd [arg]	改变目录，如果不带参数，则回到主目录，带参数则切换到参数所指的目录
command comand [arg]	即使有同名函数，仍然执行该命令。也就是说，跳过函数查找[21]
continue [n]	请参见 14.6 节“continue 命令”
declare [var]	显示所有变量，或用可选属性声明变量[20]
dirs	显示当前记录的目录(pushd 的结果)
disown	从作业表中删除一个活动作业
echo [args]	显示 args 并换行
enable	启用或禁用 shell 内置的命令[20]
eval [args]	把 args 读入 shell，并执行产生的命令

(续表)

命　令	功　能
exec command	运行命令，替换掉当前 shell
exit [n]	以状态 n 退出 shell
export [var]	使变量可被子 shell 识别
fc	历史的修改命令，用于编辑历史命令
fg	把后台作业放到前台
getopts	解析并处理命令行选项
hash	控制用于加速命令查找的内部哈希表
help [command]	显示关于内置命令的有用信息。如果指定了一个命令，则将显示该命令的详细信息[21]
history	显示带行号的命令历史列表
jobs	显示放到后台的作业
kill [-signal process]	向由 PID 号或作业号指定的进程发送信号。输入 kill –l 查看信号列表
let	用来计算算术表达式的值，并把算术运算的结果赋给变量
local	用在函数中，把变量的作用域限制在函数内部
logout	退出登录 shell
popd	从目录栈中删除项
pushd	向目录栈中增加项
pwd	打印出当前的工作目录
read [var]	从标准输入读取一行，保存到变量 var 中
readonly [var]	将变量 var 设为只读，不允许重置该变量
return [n]	从函数中退出，n 是指定给 return 命令的退出状态值
set	设置选项和位置参量。见表 14-2
shift [n]	将位置参量左移 n 次
stop pid	暂停第 pid 号进程的运行
suspend	终止当前 shell 的运行(对登录 shell 无效)
test	检查文件类型，并计算条件表达式
times	显示由当前 shell 启动的进程运行所累计用户时间和系统时间
trap [arg] [n]	当 shell 收到信号 n(n 为 0、1、2 或 15)时，执行 arg
type [command]	显示命令的类型，例如：pwd 是 shell 的一个内置命令
typeset	同 declare。设置变量并赋予其属性
ulimit	显示或设置进程可用资源的最大限额
umask [八进制数字]	用户文件关于属主、属组和其他用户的创建模式掩码
unalias	取消所有的命令别名设置
unset [name]	取消指定变量的值或函数的定义
wait [pid#n]	等待 pid 号为 n 的后台进程结束，并报告它的结束状态



14.13 bash shell 的习题

习题 54 第一个 bash shell 脚本

1. 编写一个名为 greetme 的脚本，它包括以下内容。
   a) 包含一段注释，列出您的姓名、脚本的名称和编写这个脚本的目的。
   b) 问候用户。
   c) 显示日期和时间。
   d) 显示这个月的日历。
   e) 显示您的机器名。
   f) 显示当前这个操作系统的名称和版本(cat /etc/motd)。
   g) 显示父目录中的所有文件的列表。
   h) 显示 root 正在运行的所有进程。
   i) 显示变量 TERM、PATH 和 HOME 的值。
   j) 显示磁盘使用情况(du)。
   k) 用 id 命令打印出您的组 ID。
   l) 显示“Please，couldn you loan me $50.00?”
   m) 跟用户说“Good bye”并且告诉他当前的时间(请参考 date 命令的手册页)。
2. 确保脚本可执行。

```
chomd +x greetme
```

您的脚本中第一行是什么？为什么要加这一行？

习题 55 命令行参数

1. 编写一个名为 rename 的脚本，这个脚本需要两个参数：第一个参数是文件的原名，第二个参数则是文件的新名称。

如果用户没有提供两个参数，就在屏幕上显示一条信息提示脚本的用法，然后退出脚本。下面是说明该脚本如何工作的一个例子：

```
$ rename
Usage: rename oldfilename newfilename
$

$ rename file1 file2
file1 has been renamed file2
Here is a listing of the directory:
a file2
b file.bak
```

2. 下面的 find 命令(SunOS 系统上)将列出根分区上所有大于 100KB、并且在上周被修改过的文件，(查看自己所用系统上的手册页，以确定 find 命令在当前系统上的正确语法)。

```
find / -xdev -mtime -7 -size +200 -print
```

3. 编写一个名为 bigfiles 的脚本。这个脚本带两个参数：一个是 mtime 的值，另一个则是 size 的值。如果用户没有提供两个参数，就向 stderr 发送一条合适的报错信息。

4. 编写一个名为 vib 的脚本，用它来为 vi 创建备份文件。备份文件的名称是在原始文件的名称加上后缀.bak。

习题 56 获取用户的输入

1. 编写一个名为 nosy 的脚本，该脚本将执行下列操作：
   a) 询问用户的全名——名字和姓。
   b) 用用户的名字问候他(她)。
   c) 询问用户的出生年份，并计算出他(她)的年龄(使用 let 命令)。
   d) 询问用户的登录名，并打印他(她)的用户 ID(从/etc/passwd 中获得)。
   e) 告诉用户他(她)的主目录在哪儿。
   f) 向用户显示他(她)正在运行的进程。
   g) 告诉用户现在是星期几，并且用非军用的时间格式告诉他(她)现在的时间。输出结果应类似于：

```
The day of the week is Tuesday and the current time is 04:07:38 PM.
```

2. 创建一个名为 datafile 的文本文件(除非已提供了这个文件)。文件中每条记录包含若干由冒号分隔的字段。记录中的字段包括：
   a) 名和姓
   b) 电话号码
   c) 地址
   d) 出生日期
   e) 工资
3. 创建一个名为 lookup 的脚本，让它完成如下任务：
   a) 包含一段注释，用来说明脚本名、作者姓名、时间和编写这个脚本的原因。编写这个脚本的目的是要将 datafile 的内容按顺序显示。
   b) 按姓氏对 datafile 排序。
   c) 向用户显示 datafile 的内容。
   d) 告诉用户文件中一共有多少条记录。
4. 尝试用-x 和-v 选项来调试脚本。如何使用这些命令？它们有何不同？

习题 57 条件语句

1. 编写一个名为 checking 的脚本来执行如下操作：
   a) 接收一个命令行参数：用户的登录名。
   b) 检查用户是否提供了命令行参数。
   c) 检查用户是否在/etc/passwd 文件中，如果在，就显示信息：

```
Found <user> in the /etc/passwd file.
```

若不在，则显示信息：



```
No such user on our system.
```

2. 在脚本 lookup 中，询问用户是否要往文件 datafile 增加一条记录。如果用户回答 yes 或 y，则：

a) 提示用户输入姓名、电话号码、地址、出生日期和工资。将每一项分别保存在一个单独的变量中。在字段间加冒号，然后把这条信息追加到文件 datafile 尾部。

b) 按姓氏对该文件排序。告诉用户这条记录已被加入，向他(她)显示该行，并在行首标出行号。

习题 58　条件语句与文件测试

1. 改写 checking 脚本。检查完指定的用户是否在/etc/passwd 文件中之后，程序接着检查这个用户是否已登入系统。如果是，程序就打印出正在运行的所有进程。否则，程序将告诉用户：

```
<所指定的用户> is not logged on.
```

2. 用 let 命令计算一组等级。脚本请用户输入他(她)在一次考试中的分数(使用 declare －i)，然后测试用户输入的分数是否在有效范围 0～100 之内。如果不在，则程序退出。如果在，则显示用户的等级字母，如：You receive an A. Excellent！不同等级的分数范围：A(90－100) B(80－89) C(70－79) D(60－69) F(小于 60)

3. 脚本 lookup 要依靠 datafile 文件才能运行。在脚本 lookup 中，检查文件 datafile 是否存在，是否可读且可写。在脚本 lookup 中增加一个如下所示的菜单。

```
[1] Add entry
[2] Delete entry
[3] View entry
[4] Exit
```

您已经在脚本中写了增加条目(Add entry)的部分。现在要在增加条目的程序中添加代码，以检查用户提供的姓名是否已在文件 datafile 中出现。如果在，就告诉用户。如果不在，则增加这条新记录。

现在编写删除条目(Delete entry)、查看条目(View entry)和退出(Exit)函数的代码。

脚本中处理删除的部分应该首先检查条目是否存在，然后才去删除它。如果条目不存在，则向用户报告错误。如果存在，就删除它，并且告诉用户条目已被删除。退出时，一定要用一个数字来代表适当的退出状态。

如何从命令行检查退出状态？

习题 59　case 语句

1. BSD Berkeley UNIX 上的 ps 命令与 System 5(AT&T UNIX)以及 Linux 上的有所不同。BSD Unix 的 ps 命令使用 BSD 选项。在 System 5 上，列出所有进程的命令是：

```
ps -ef
```

而 BSD UNIX 上对应的命令则是：

```
ps -aux
```

请您编写一个名为 systype 的程序，用它来检查各种不同的系统类型。要测试的系统将包括：

```
AIX
Darwin(Mac OS X)
Free BSD
HP-UX
IRIX
Linux
OS
OSF1
SCO
SunOS(Solaris/SunOS)
ULTRIX
```

Solaris、HP-UX、SCO 和 IRIX 是 AT&T 一类的系统，其余的则是 BSD 风格的系统。

您正在使用的 UNIX 的版本信息将被打印到 stdout。系统的名称可以用 uname -s 命令或从文件/etc/motd 中获得。

2. 编写一个名为 timegreet 的脚本，完成下面的任务：

a) 在脚本顶部编写一个注释段，说明作者姓名、日期和程序的目的。

b) 把下面的用 if/elif 实现的程序转换为用 case 命令来实现。

```
#!/bin/bash
# Conmment section
you=$(date +%H)
echo "The time is: $(date +%T)"
if ((hour > 0 && hour <12))
then
  echo "Lnuch time!"
elif ((hour > 12 && hour <16))
then
  echo "Good afternoon, $you!"
else
  echo "Good noght, $you. Sweet dreams."
fi
```

习题 60　循环

选做一题：

1. 编写一个名为 mchecker 的脚本，用来检查是否有新邮件到达，如果有新邮件，则在屏幕上显示一条消息。

a) 程序取得用户的邮件假脱机文件的长度(AT&T 类系统上，邮件假脱机文件的位置是/usr/mail/$LOGNAME 中，UNIX 类系统上，假脱机文件的位置是在/usr/spool/mail/$USER 中。如果您找不到它们，可以使用 find 命令)。这个脚本将连续循环运行，每 30 秒一次。每一轮循环时，脚本都将邮件假脱机文件的长度与上一轮循环时取得的长度进行比较。如果新的长度大于旧的长度，就在屏幕上显示一条消息：<用户名>，You have new mail.



文件的长度可以从 ls －l，wc －c 和 find 命令的输出中找到。

2. 编写一个脚本完成下面的任务。

a) 在脚本顶部编写一个注释段，说明作者姓名、日期和编写程序的目的。

b) 使用 select 循环生成一个食品菜单。

c) 程序的输出如下所示：

```
1)  steak and potatoes
2)  fish and chips
3) soup and salad
Please make a selection. 1
Stick to your ribs.
Watch your cholesterol.
Enjoy your meal.
1)  steak and potatoes
2)  fish and chips
3) soup and salad
Please make a selection. 2
British are coming!
Enjoy your meal.
1)  steak and potatoes
2) fish and chips
3) soup and salad
Please make a selection. 3
Health foods...
Dieting is so boring.
Enjoy your meal.
```

3. 编写一个名为 dusage 的脚本程序，向一组用户逐一发送邮件，告诉用户他(她)当前已用的磁盘块数目。用户名列表保存在文件 potential_hogs 中。potential_hogs 存在一个用户名为 admin。

a) 用文件测试检查文件 potential_hogs 是否存在并可读。

b) 用循环遍历整个用户名列表。只向磁盘用量超过 500 块的用户发送邮件。跳过用户 admin，不向他(她)发邮件，也就是说，admin 用户不会收到该邮件信息。邮件的信息保存在 dusage 脚本的 here 文档中。

c) 保存一份收到邮件的用户的名单。通过创建日志文件来完成这一目标。给用户列表上所有用户都发送完邮件后，打印收到邮件的人数及名单。

习题 61　函数

1. 将实习题 59 中的 systype 程序改写为一个返回系统名的函数。调用该函数来确定使用 ps 命令时应使用哪些选项。

在 AT&T UNIX 上列出所有进程的 ps 命令为：

```
ps -ef
```

而在 UNIX/BSD UNIX/Linux 上，命令为：

```
ps -aux or ps aux
```
[22]

2. 编写一个名为 cleanup 的函数，它将删除所有临时文件并退出脚本。如果程序运行过程中收到终端或挂起信号，trap 命令将调用 cleanup 函数。

3. 用 here 文档在脚本 lookup 中增加一个如下所示的菜单。

```
[1] Add entry
[2] Delete entry
[3] Change entry
[4] View entry
[5] Exit
```

为每个菜单项编写一个处理函数。用户选择一个有效菜单项之后，程序执行对应的函数，然后询问用户是否想再看一遍菜单。如果用户输入菜单项无效，则显示信息：

```
Invalid entry, try again.
```

然后重新显示菜单。

4. 在 lookup 脚本的菜单项 View entry 下创建一个子菜单。询问用户是否要看所选的那个人的某项特定信息：

a) Phone

b) Address

c) Birthday

d) Salary

5. 在脚本使用 trap 命令，使程序在运行过程中收到中断信号时，以执行清除操作。

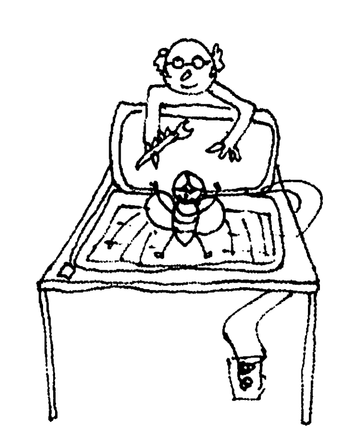
[22] UNIX 系统上，如果在前面加了短划线，会收到警告信息。请参考手册页。不带短划线时，Linux 显示的是用户的进程。



chapter

15

调试 shell 脚本



15.1 简介

通常程序员在调试程序上所耗费的时间要大于编写程序所用的时间。因为许多 shell 程序执行的任务会影响到整个操作系统，所以必须保证这些脚本能够正确执行。无论是用户还是系统管理员，他们通过编写脚本以自动执行一些简单或复杂的任务时，都希望脚本能够按照自己的意图行事，在清除了所有的语法错误之后不会再出现任何怪异的或意料之外的问题。当读者想通过某种方式对已经可以正常工作的脚本进行改进时，他将忽视“除非万不得已，决不轻言改动”的传统约定。或许脚本界面应该更加友好，可能需要一次“美容手术”。或许需要加入更多的错误检查；或许需要清理一下其他程序员的代码。但是，当您将脚本再次置于编辑器中并大动干戈，费了九牛二虎之力后，退出编辑器且运行程序时，您很可能会发现程序垮掉了。因此，本章致力于提供一个这样的工具，它可以发现、修复和理解各种可能导致脚本行为异常的错误，这些错误往往会花费程序员很多时间来调试，甚至可能超出他们的可支配时间。

15.2 风格问题

尽管不是必需的，但程序风格的好坏对发现程序中的 bug 意义重大。下面给出了一些简单的指导原则。

- 在程序中加入注释以使自己和他人都能够明白程序的用途。在注释上偷工减料会为今后的调试埋下重大隐患。
- 定义有意义的变量名，把它们放在程序头部以利于发现拼写错误和空值。因为类似 foo1, foo2 和 foo3 之类的名称不能直观地显示任何信息。确保变量名没有使用保留字并注意大小写问题。



- 无论何时使用条件命令或循环命令，将后续的语句至少缩进一个制表符的距离。如果存在条件或循环嵌套，缩进会更多。对齐条件或循环命令和它们的结束关键字，如 if 和 endif，if 和 fi，while 和 done 等(参见 15.4.2 节中“遗漏的关键字”和“缩进”)。
- 在一直出现语法错误的地方使用 echo 命令或打开 echoing 和 verbose 开关以跟踪程序的执行(参见 15.5 节“使用 shell 选项和 set 命令进行跟踪”)。
- 即使程序可以无错误地运行，可能还会有一些潜在的逻辑错误。熟悉运算符的使用，它们因 shell 而异，经常会引起逻辑错误。
- 确保程序是健壮的。检查所有可能的人为错误，例如输入错误、参数不足、文件不存在等类似的问题(参见 15.4.4 节“逻辑错误与健壮性”)。
- 使用简短的测试内容。例如，测试一个函数的语法，只需用一个小的脚本测试，看结果是否与预期一致。
- 了解操作系统命令。因为大多数语句是由 UNIX/Linux 命令组成的，如果要在脚本中使用一个命令但又不确定它的行为，可以先在命令行使用这个命令。看是否了解它的退出码，变量是如何解释的，如何正确地引用，如何重定向输出和错误?
- 最后，如果您是系统管理员，在系统上应用脚本前，应仔细测试。一个意外的错误可能会导致整个系统崩溃。

下面的指导原则将帮助您解决上述的所有问题，并给出各 shell 最常见错误的列表，引起错误的原因以及如何修复。

15.3 错误类型

shell 程序可以导致下面几种类型的错误：运行时错误、逻辑错误和程序健壮性错误(如果程序能够满足健壮性要求，则能够避免这种类型的错误)。运行时错误一般发生在启动脚本运行后遇到一些意外情况，如使用了错误的脚本名、权限问题、路径问题或语法错误，譬如引用不匹配或词拼写错误。如果发生这种类型的错误，shell 将报告一些诊断信息并终止程序运行。逻辑错误通常更难以发现，因为它们并不是显而易见的。这种类型的错误甚至与程序构造的方式有关，如在表达式中使用了错误的运算符、不了解命令的退出状态值或使用了有缺陷的分支或循环语句。其他的错误主要是因为程序缺乏健壮性，也就是说，这些错误本应该通过合适的错误检查加以避免。这类的错误包括出现未能预见的用户输入或参数不足等问题。

15.3.1 运行时错误

运行时错误一般发生在脚本开始运行后遇到一些意外情况，如使用了错误的脚本名、权限问题、路径问题或语法错误，如引用不匹配或词拼写错误。

15.3.2 命名惯例

在 shell 提示符处键入一个命令，shell 将搜索 UNIX/Linux 路径以查找这条命令。当在

提示符处键入一个脚本名时，它被当作另一个命令看待。shell 将搜索路径以寻找它。现在假定你将一个脚本命名为某个 UNIX 命令，例如，将程序命名为“ls”。哪个 ls 将被执行？这取决于在搜索路径中先找到的是哪一个，很可能执行的是 UNIX 命令而非您的脚本。很显然脚本不应该被命名为 ls 或 cat，但对一些类似 test 或 script 的不常用的命令就不是那么明显了。例如，因为 UNIX 的 test 命令并不产生输出，所以命名问题比较复杂。UNIX 的 script 命令启动一个新的交互式 shell，它将您在当前终端的会话复制至 typescript 文件。新的 shell 启动运行并不是显而易见的，往往在你意识到出错之前它已经运行了一段时间。所以，不但要注意不要将文件命名为 cat 或 ls 这些常用的命令，更要注意避免将其命名为 test 或 script 这些不常用的命令。

使用 which 命令可以知道脚本是否与 UNIX/Linux 命令同名。它将显示找到的命名程序的路径，参见范例 15-1。也可以以./启动脚本，这样当前工作目录中的脚本将会被运行。

范例 15-1

```
1   $ cat test
    #! /bin/sh
    echo "She sells C shells on the C shore"
2   $ test
    $ (no output)
3   $ echo $PATH
    /usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin:/home/ellie/bin:.
4   $ which test
    /usr/bin/test
5   $ mv test mytest
6   $ mytest
    She sells C shells on the C shore
```

说明

1. 注意，脚本名为 test，这同时也是一个 UNIX/Linux 命令名。

2. 当用户希望运行 test 脚本时，实际运行的是 UNIX/Linux 命令。UNIX 的 test 命令并不产生输出。

3. 显示了用户搜索路径。首先被搜索的路径是/usr/bin，路径结尾的点代表用户当前工作目录。

4. which 命令定位 test 命令并显示了它的完整路径名。UNIX/Linux test 命令位于/usr/bin/test。

5，6. 如果 test 脚本能够被命名为一个唯一的名字，而不与命令同名，则它将按预期执行。test 脚本被改名为 mytest。

15.3.3 参数不足

普通文件没有执行权限。如果直接从命令行运行脚本，则必须将它的执行权限打开，否则 shell 将提示没有足够的权限。可以使用 chmod 命令将脚本置为可执行的。

范例 15-2

```
1   $ mytest
```



```
    sh: ./mytest: Permission denied.
2   $ ls -l mytest
    -rw-rw-r-- 1  ellie   ellie   23 Jan  22   12:37   mytest
3   $ chmod +x mytest    # or chmod 755 mytest
4   $ ls -l mytest
    -rwxrwxr-x 1  ellie   ellie   23 Jan  22   12:37   mytest
5   $ mytest
She sells C shells on the C shore
```

说明

1. 执行脚本时，错误信息指出脚本没有合适的执行权限。
2. ls -l 命令显示任何人都没有该脚本的执行权限，包括脚本的所有者。
3. chomd 命令为所有人增加执行权限(+x)。现在脚本就可以运行了。
4. ls -l 命令显示了新的权限。执行权限被打开。
5. 脚本按预期执行。

15.3.4 路径问题

之前曾提到过，在命令行键入脚本名，shell 将检查列于搜索 PATH 变量中的目录，以定位此脚本。如果路径中列有点(.)目录，则 shell 将在当前工作目录中搜索脚本。如果脚本在当前目录下，那么它将被执行。但是，如果点(.)目录不在搜索路径上，则将收到错误信息：Command not found。为使 shell 能够定位脚本，可以将点目录(.)加入到 PATH。然而，如果您拥有超级用户账户(UID 为 0 的账户)，为安全起见，建议不把它加入到 PATH 中。如果选择不将点目录放到 PATH 变量中，还有其他两种可选方案：

(1) 在脚本名前加上一个./以显式地指出脚本位置(例如，./mytest)

(2) 在脚本名前加上调用 shell 的名称(例如，csh mytest 或 bash mytest)

如果脚本名作为一个参数传给 shell，shell 将自动在当前目录中查找该脚本。

范例 15-3

```
1   $ cat mytest
    #!/bin/sh
    echo "Users may wish to include the . directory in their PATH"
2   $ echo $PATH
    /usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin:/home/ellie/bin:
3   $ mytest
    not found
4   $ ./mytest      # this is one possible solution
    Users may wish to include the . directory in their PATH
5   $ PATH=$PATH:.  # this solution should only be used by non-root
    $ export PATH   # bash, sh, ksh set PATH this way
    % setenv PATH ${PATH}:.   # csh/tcsh set path this way
6   $ mytest
    Users may wish to include the . directory in their PATH
```

说明

1. 显示 Bourne shell 脚本的内容。它位于用户当前工作目录。

2. PATH 变量不包含代表当前工作目录的点目录。

3. shell 不能够定位 mytest 脚本。

4. shell 能够定位脚本，因为脚本名中包含有点目录(.)。

5. PATH 变量被更新以加入当前目录。sh/ksh/bash shell 使用第一种语法，csh/tcsh 使用第二种语法。

6. 脚本正确运行。

15.3.5 shbang 行

创建脚本时，脚本的第一行通常称为 shbang(#!)行。当脚本启动后，UNIX 内核检查文件的第一行以决定将要执行的程序类型。shbang 符号(#!)后面的路径是用来指定解释此脚本的 shell 的位置。要正确使用这个特性，#! 必须是文件中最前面的两个字符。如果文件头部有空白行或空格字符，则此特性被忽略，该行被解释为普通的注释行。

范例 15-4

```
(脚本)
1
2   #!/bin/csh
    # Scriptname: shbang.test
3   setenv MYVAR 18
    echo "The value of MYVAR is $MYVAR

(输出)
    ./shbang.test: line 3: setenv: command not found
    The value of MYVAR is

(脚本)
4   #!/bin/csh
    setenv MYVAR 18
    echo "The value of MYVAR is $MYVAR

(输出)
The value of MYVAR is 18
```

说明

1. 注意脚本的头部有一个空白行，同时脚本使用 C shell。而第 3 行的 setenv 命令在 Bourne，Korn 或 bash shell 中不起作用。

2，3. 因为#!行不是脚本的第一行，因此脚本将由用户登录 shell 执行。如果登录 shell 为 ksh，则 Korn shell 将执行这个脚本；如果登录 shell 为 bash，则 bash shell 将执行该脚本。然而，如果登录 shell 为 C shell，则脚本将使用 bash shell 执行。显示出的错误信息是由于 Bourne shell 不理解 setenv 命令造成的。

4. 脚本正常运行，这是因为#! 行被置为首行。内核将识别出#! 并启动它后面的路径中列出的 shell。因为 shell 为/bin/csh，因此 C shell 将作为此脚本(执行权限被打开)的解释器。C shell 能够解释 setenv 命令。



范例 15-5

```
1   #! /bin/chs
    echo "Watch out for typing errors on the #! line"
2   $ ./mytest
    mytest: Command not found.

-----------------------------------------

3   #! /bin/csh
    echo "Watch out for typing errors on the #! line"
4   ./mytest
    Watch out for typing errors on the #! line
```

说明

1. shbang 行的 shell 名存在键入错误。shell 应该为/bin/csh 而不是/bin/chs。shbang 行是经常容易出现拼写错误的地方。

2. 因为存在键入错误，内核查找/bin/chs 程序，从而导致 Command not found 错误信息。这个信息可能会被误解，以为是脚本自身或脚本中键入的命令不能被找到。如果您仅得到 Command not found 这一个错误，首先应检查 shbang 行的拼写。

3. shell 中 shbang 行的拼写错误被更正。

4. 脚本可以正常运行。

15.3.6 别名问题

别名是命令的缩写或另一个的名称。Bourne shell 不支持别名。别名通常放在初始化文件里，在命令行用作快捷方式，但很少在脚本中使用。然而，如果用户的.cshrc 或.kshrc 文件定义了别名但却没有将其限制在交互式 shell 会话中，则别名也可能在脚本中使用。

范例 15-6

```
(.cshrc 文件中)
1   alias ls 'ls -aF'

(shell 提示符处)
2   $ ls
    ./      a       c       t1      t3
    ../     b       t*      t2      tester*

(脚本)
    #!/bin/csh
3   foreach i ( `ls` )
4       echo $i
    end

(输出)
./
../
a
```

```
b
c
t t1 t2 t3 tester   <-- What happened here?
t1
t2
t3
tester

----------------Possible Fix--------------------------

5  #!/bin/csh -f     <-- Add the -f option to the shbang line
6  unalias ls        <-- turn off the alias
```

说明

1. 在用户的初始化文件.cshrc 中，定义了一个别名。通常任何时候调用一个 C shell 或者一个 C shell 脚本，.cshrc 文件都将被启动。一般.cshrc 文件在表达式后定义别名：

```
if ( $?prompt ) then
  < aliases listed here >
endif
```

这意味着：如果运行一个交互式 shell，即出现提示符的 shell，就可以使用这些别名。因为 shell 脚本是非交互的并且没有提示符，所以它们不能使用别名(别名通常也存放在文件.aliases 中，从文件.cshrc、.kshrc、.bashrc 或.tcshrc 中可以得到)。

2. 当用户在提示符处键入 ls 时，就使用了别名。它将导致 ls –aF 被执行。-F 开关对可执行文件添加一个*号，为目录文件添加一个/号，为符号连接添加一个@号。-a 开关打印出隐藏文件(也就是以句点开头的文件)。

3. 在 shell 脚本中，ls 命令在 foreach 表达式中执行。shell 将对别名求值，从而 ls –aF 命令将按上面提到的格式列出文件，结果有些出乎意料。每个可执行文件被添加了一个*号，shell 将其看作是元字符并进行文件名匹配。因为文件 t 是一个可执行文件，它被列为 a*，从而导致 shell 匹配所有以 t 开头的文件。

4. 用别名列出所有的文件。

5. 这个问题的一个解决方法是为 shbang 行的/bin/csh 增加-f 选项。通常称作快速启动选项，-f 开关通知 shell 启动脚本时不加载.cshrc 文件。对 ksh 其 shbang 行可能为#!/bin/ksh –p，对 bash 可能为#!/bin/bash --noprofile。

15.4 可能导致语法错误的原因

15.4.1 未定义变量与误写变量

众所周知，shell 使用变量时并不需要预先进行声明。当在程序中使用一个变量名时，该变量被自动创建。尽管看起来这种方式比显式地定义每个变量更加方便，但它却同时带来一个不良的后果：一个不经意的拼写错误可能会创建本来无须创建的多余变量。又因为

UNIX/Linux 是区分大小写的，即使只是将大写变成了小写也可能导致程序失效。

当在 C/TC shell 中设置变量时，将要用到 set 命令，并且符号=必须前后都有空格(或都没有)。bash、sh 和 ksh 中的 set 命令用于设置 shell 选项或创建位置参量，但它不用于定义变量。符号=前后都不允许出现空格。在各种 shell 之间切换时，很容易忘记应何时使用 set 命令，何时使用或不使用空格。

如果使用了未定义的变量，C shell 与 TC shell 将给出通知。除非使用 set –u 或 shell 的 -u 调用选项要求进行错误检查，否则 Bourne、Bash 和 Korn shell 将显示一个空行。-u 选项标志着未定义的变量，称为无约束变量。

在 Bourne、Korn 和 Bash shell 中，变量以如下方式设置：

```
x=5
name=John
friend="John Doe"
empty=   or empty=""
```

检查未定义变量：

```
set -u
echo "Hi $firstname"

(输出)
ksh: firstname: parameter not set
```

在 C shell 与 TC shell 中：

```
set x = 5
set name = John
set friend = "John Doe"
set empty = ""
```

shell 检查未定义变量：

```
echo "Hi $firstname"

(输出)
firstname: Undefined variable
```

范例 15-7

```
#!/bin/tcsh
1   set friend1 = "George"
    set friend2 = "Jake"
    set friend4 = "Danny"
2   echo "Welcome $friend3 "

(输出)
3   friend3: Undefined variable.
```

说明

1. 设置 3 个变量。如果变量名相似，则在以后的程序中很容易弄错。

2. 变量 friend3 未定义。

3. C shell 与 TC shell 会发送一个错误从而使你明白使用了未定义的变量。Bourne、Korn 和 bash 则仅显示一个空行。

15.4.2 未完成的编程语句

遗漏关键字 当使用编程语句如 if 语句或 while 循环时，可能会不小心遗漏了部分语句。例如，使用 while 循环时，可能会忘记加上结尾的关键字 done。只要错误信息提到 unexcepted end of file 或者出错的行刚好是最后一行语句的下一行(例如，脚本包含 10 行且消息指出第 11 行出错)，则应该检查是否有未完成的编程语句。

缩进 为确保 if/else、while/do/done、case 语句及其他结构的完整性，一种简单的方式是将这些语句块在表达式下缩进(至少一个 tab)，同时将终止关键字(done、end、endsw 等)与它所终止的条件或循环命令对齐。在这些结构嵌套中时，使用缩进的好处尤其明显。

```
在 sh,bash,ksh 中:
  if [ expression ]
  then
    statement
    statement
  fi
```

```
在 csh,tcsh 中:
  if ( expression ) then
     statement
     statement
  endif
```

if/endif 错误 参照下面的格式正确地使用 tab 键来设置 if/endif 结构。

格式

```
在 Bourne,Korn 和 Bash shell 中:
# Without indentation
if [ $ch = "a" ]    # Use indentation
then
echo $ch
if [ $ch = "b" ]    <-- Missing 'then'
echo $ch
else
echo $ch
fi
       <-- Missing 'fi' for first 'fi'
--------------------------Fix--------------------------
# With indentation
if [ $ch = "a" ]
then
   echo $ch
   if [ $ch = "b" ]
   then
      echo $ch
   else   # 'else'goes with nearest 'if'
      echo $ch
   fi
fi
在 C shell 与 TC shell 中:
```




```
if ( $ch == "a" )      <-- Missing 'then'
    echo $ch
    if ( $ch == "b" ) then
        echo $ch
else
    echo $ch
endif
      <--Missing 'endif' for first 'if'
--------------------------Fix--------------------------
if ( $ch == "a" ) then
    echo $ch
    if ( $ch == "b" ) then
        echo $ch
    else
        echo $ch
    endif
endif
```

case 与 switch 错误 case 与 switch 命令中常常会隐藏一些 bug。被测试的变量如果包含的词多于一个时，它应该用双引号引起来。关系、逻辑和相等运算符不能出现在 case 常量中。

格式

```
Bourne,Korn 和 Bash shell 中的 case 语句:
case $color in          <-- Variable should be quoted
blue)
   statements
   statements           <-- Missing ;;
red || orange)          <-- Logical || not allowed
   statements
     ;;
*)  statements
     ;;
                        <-- Missing esac
-----------------------The Fix--------------------------------
case "$color" in
blue )
   statement
   statement
     ;;
red | orange )
   statements
     ;;
*)
   statements
     ;;
esac

C shell 与 TC shell 中的 switch 语句:
switch ($color)          <-- Variable should me quoted
```

```
case blue:
   statements
   statements           <-- Missing breaksw
case red || orange:     <--Logical operator not allowed
   statements
   breaksw
default:
   statements
   breaksw
                        <-- Missing endsw
-----------------------The Fix--------------------------------
switch ("$color")
case blue:
   statements
   statements
   breaksw
case red:
case orange:
   statements
   breaksw
default:
   statements
   breaksw
endsw
```

循环错误 循环错误是指 for、foreach、while 或 until 循环的语法不正确，多数情况下是用于终止循环块的关键字如 do、done 或 end 被遗漏了。

格式

```
Bourne shell:
while [ $n -lt 10 ]     <-- Missing do keyword
   echo $n
   n=`expr $n + 1`
done

while [ $n -lt 10 ]
do
   echo $n
   n=`expr $n + 1`
                       <-- Missing done keyword
-----------------------The Fix--------------------------------
 while [ $n -lt 10 ]
do
   echo $n
   n=`expr $n + 1`
done
Bash shell 与 Korn shell 中的循环:
while (( $n <= 10 ))   <-- Missing do keyword
   echo $n
   (( n+=1 ))
done
```



```
while  (( $n <= 10 ))
do
   echo $n
   (( n+=1 ))
                              <-- Missing done keyword
-----------------------The Fix---------------------------------
while  (( $n <= 10 ))
do
   echo $n
   (( n+=1 ))
done
```

C shell 与 TC shell 中的循环：

```
while ( $n <= 10 )
   echo $n
   @n+=1                        <-- Missing space after the @ symbol
                                <-- Missing end keyword
foreach ( a b c )               <-- Missing variable after foreach
   echo $char
end
-----------------------The Fix---------------------------------
while ( $n <= 10 )
   echo $n
   @ n+=1
end

foreach char ( a b c )
   echo $char
end
```

运算符错误　各种 shell 用于操作字符串和数字的操作符互不相同。Bourne shell 使用 test 命令(参见 test 帮助)及其操作符来比较数字和字符串。尽管 Korn shell 和 bash shell 也有这些操作符，但通常并不使用它们，而是提供一个类 C 的操作符集合，使用 let 命令(( ))来处理算术运算，而使用新的 test 命令[[ ]]来处理字符串操作。不同的是，bash shell 使用双等号(= =)表示相等关系，而 Korn shell 则不使用。

C shell 与 TC shell 也提供了一个类 C 的操作符集合以比较数字和字符串，并对两者都使用双等号==。是不是有些混乱？如果您要移植 shell 脚本，最好检查一下每种 shell 的操作符。本书为每种 shell 的操作符都进行了列表说明(参见附录 B)。下面的例子将会解释每种 shell 的一些操作符的用法。

格式

Bourne shell:

数字测试

```
   if [ $n -lt 10 ]
   if [ $n -gt $y ]
   if [ $n -eq 6 ]
   if [ $n -ne 6 ]
```

字符串测试

```
   if [ "$name" = "John" ]
   if [ "$name" != "John" ]
```

Korn shell:
数字测试
```
   if (( n < 10 ))
   if (( n >  y ))
   if (( n == 6 ))
   if (( n != 6 ))
```
字符串测试
```
   if [[ $name = "John" ]]
   if [[ $name != "John" ]]
```

Bash shell:
数字测试
```
   if (( n < 10 ))
   if (( n >  y ))
   if (( n == 6 ))
   if (( n != 6 ))
```
字符串测试
```
   if [[ $n == "John" ]]
   if [[ $n != "John" ]]
```

C/TC shell:
数字测试
```
   if ( $n < 10 )
   if ( $n >  $y )
   if ( n == 6 )
   if ( n != 6 )
```

字符串测试
```
   if ( "$name" == "John" )
   if ( "$name" != "John" )
```
误用操作符 下面的例子解释了最常见的误用操作符的情况。

范例 15-8

```
(sh)
1  n=5; name="Tom"
2  if [ $n > 0 ]    # Shold be: if [ $n -gt 0 ]
   then
3  if [ $n == 5 ]   # Shold be: if [ $n -eq 5 ]
   then
4  n++     # Shold be: n=`expr $n + 1`
5  if [ "$name" == "Tom" ]      # Shold be: if [ $name = "Tom" ]

(csh/tcsh)
   set n = 5; set name = "Tom"
6  if ( $n =< 5 ) then           # Shold be: if ( $n <= 5 ) then
7  if ( $n == 5 && < 6 ) then   # Shold be: if ( $n == 5 && $n < 6)
8  if ( $name == [Tt]om ) then  # Shold be: if ($name =~ [Tt]om )
```




```
(ksh)
   name="Tom"
   n=5
9  if [ $name == [Tt]om ]      # Shold be: if [[ $name == [Tt]om ]]①
[[ n+=5 ]]  # Should be: (( n+=5 ))
```

说明

1. 在 Bourne shell 中，变量 n 被赋值为 5。

2. 方括号[是 test 命令的符号。test 命令不使用>来表示大于，而是使用-gt 表示关系操作符。

3. 双等号不是 test 命令有效的相等操作符。正确的操作符应该是-eq。

4. Bourne shell 不支持算术运算。

5. test 命令不使用= =进行相等测试；它对字符串测试使用单个等号，对数字测试使用-eq。

6. csh/tcsh 关系操作符应该为<=。在所有 shell 中误用关系操作符将导致语法错误。

7. 逻辑操作符&&右侧的表达式是不完整的。

8. csh/tcsh shell 使用=~对包含通配的字符串求值。本例中的错误应该为 No match。

9. 单独一对方括号类型的测试不支持= =符号。bash 和 ksh 使用[[test 命令测试包含通配符的表达式。使用双等于号，而不是单个等于号来测试字符串是否相等(仅比 ksh88 版本新的 ksh 支持==符号，早期版本只需使用单个=号来测试字符串)。

10. [[test 命令用于字符串表达式。((let 命令用于数字表达式。

引用错误 误引用是 shell 脚本中常见的一种错误，为此这里我们使用一整节来描述引用问题(参见稍后的“您不可不知的引用”)。引用通常称作“shell 程序员的克星”或“来自地狱的引用”。有 3 种类型的引用：单引用、双引用和反引用。单引用和双引用一般引用文本字符串以防止它们被解释，反引用用于命令替换。尽管在前面的章节中已经讨论过引用，下面几节将集中讨论如何正确使用它们以避免错误。

引用元字符 与引用直接相关的一个问题是误用元字符。我们之前已经讨论过两种类型的元字符：shell 元字符和 vi、grep、sed、awd 以及其他程序使用的正则表达式元字符(参见第 3 章“正则表达式与模式匹配”)。未引用的元字符将被 shell 解释为文件匹配，未引用的正则表达式元字符可能导致 grep、sed 和 nawk 之类的程序执行出错。下面的例子中，grep 使用*来表示 0 个或多个字母 b，并且*被 shell 用来匹配所有以“f”开头的文件名。shell 总是在执行命令之前对命令行求值。因为 grep 的*未被引用，所以 shell 将试图对它求值，从而导致一个错误。

```
grep ab*c  f*
```

为解决这个问题，必须这样进行引用：

```
grep 'ab*c'  f*
```

现在 shell 分析命令行时将不会对引号中的字符求值。

---

① 比 ksk 88 版本新的 ksh 版本，“==”符允许使用。而之前的版本中仅允许使用单个“=”符。

引用必须成对出现。大多数的 shell 在发现有不匹配的引号时都会报错。然而，Bourne shell 总是在分析完整个脚本文件后才会报告问题，通常是"unexcepted end of file"，当针对其他类型的问题也报告这个同样的错误信息时，这个消息本身将对发现问题毫无意义。

要想真正弄清楚 shell 脚本，必须对引用机制有很好的理解。

范例 15-9

```
#! /bin/csh
1   echo I don't understand you.   # Unmatched single quote

(输出)
2   Unmatched '
----------------------------------------
    #! /bin/csh
3   echo Gotta light?   # Unprotected wildcard

(输出)
4   echo: No match
----------------------------------
    #!/bin/csh
5   set name = "Steve"
6   echo 'Hello $name.'     # Variable not interpreted

(输出)
Hello $name
```

说明

1．引用必须匹配。don't 中的单个引号将导致一个错误。为解决这个问题，该字符串可用双引号引起来或在该单个引号前加一个反斜线，也就是 don\' t

2．C shell 为不匹配的单个引号显示错误信息。

3．shell 元字符?用来在文件名中匹配单个字符。目录中没有一个名为 lignt 后跟单个字符的文件。

4．C shell 报告该元字符没有匹配。为解决这个问题，或者字符串用单引号或多引号引起来，或者在问号前加上一个反斜线如'Gotta light?'或 light\?。

5．字符串"steve"被赋值给一个变量。

6．字符串以单引号引用。单引号中的字符只作字面理解(也就是，变量将不会被解释)。为解决这个问题，字符串必须以双引号引用或不引用，如"Hello $name."。

不可小看的引用　引用是 shell 脚本内部的一部分，因此专门设置本节来阐明它在所有 shell 中的用法。如果您经常遇到引用方面的语法错误，学习本节之后将确保您能够熟悉如何使用它们，特别是当脚本中包含有类似 grep、sed 和 awk 的命令时。

反斜线

1. 位于一个字符之前，转义该字符。
2. 与将一个字符放在单引号之内相同。

单引号

1．必须匹配。



2. 保护除以下之外的所有元字符免被解释。
   a. 自身
   b. 感叹号(仅 csh 适用)
   c. 反斜线

表 15-1　正确使用单引号的例子

C/TC Shell	Bourne/Bash Shell	Korn Shell
echo '$*&><?' $*&><?	echo '$*&!><?' $*&!><?	print '$*&!><?' $*&!><?
(C) echo 'I need $5.00\!' I need $5.00! (TC) echo 'I need $5.00! '	echo 'I need $5.00!' I need $5.00!	print 'I need $5.00! '
echo 'she cried,"help"' She cried,"Help"	echo ' she cried, "Help"' She cried, "Help"	print 'she cried, "Help"' She cried, "Help"
echo '\\\'	echo '\\\\'	print '\\\\'
\\\\	(Bourne) \\ (Bash) \\\\	\\

双引号

1. 必须匹配。
2. 保护除以下之外的所有元字符免被解释。
   a. 自身
   b. 感叹号(仅 csh 适用)
   c. 用于变量替换的$
   d. 用于命令替换的反引号` `

表 15-2　正确使用双引号的例子

C Shell	Bourne Shell	Korn Shell
echo "Hello $LOGNAME\!"	echo "Hello $LOGNAME!"	print "Hello $LOGNAME!"
echo "I don't care"	echo "I don't care"	print "I don't care"
echo "The date is 'date'"	echo "The date is 'date'"	print "The date is $(date)"
echo "\\\\"	echo "\\\\"	print "\\\\"
\\\\	\	\

反引号

shell 程序使用反引号进行命令替换。它们与单引号和双引号无关，但通常是问题之源。例如，当复制一个 shell 脚本时，如果用单引号替换了反引号，则程序将不会工作[②]。

② 在这种类型的书中，很容易将反引号误当作单引号。

范例 15-10

```
#!/bin/sh
1   now=`date`
2   echo Today is $now
3   echo "You have `ls|wc -l` files in this directory"
4   echo 'You have `ls|wc -l` files in this directory'
```

(输出)

```
2   Today is Mon Jul 5 10:24:06 PST 2004
3   You have 33 files in this directory
4   You have `ls|wc -l` files in this directory
```

说明

1. 将 UNIX/Linux date 命令的输出赋值给变量 now(对 T/TC shell 是: set now = `date`)。反引号导致命令替换。反引号一般位于键盘上的代字符号键(~)上。

2. 显示了变量 now 的值，也就是当前日期。

3. 对 UNIX/Linux 管道使用反引号。ls 的输出通过管道成为 wc -l 的输入。结果是统计当前目录下文件的个数。双引号中的字符串将不会进行命令替换。输出嵌入到字符串中并被打印。

4. 以单引号引用字符串，则其中的反引号将不会被解释，只作字面理解。

混合使用引号

混合使用引号是一个很有难度的问题。下一节将教您如何正确使用引号的步骤。我们将演示如何将一个 shell 变量嵌入到 awk 命令行中并在不涉及 awk 字段指示符$1 和$2 的情况下使 shell 对变量进行扩展。

设置 shell 变量

```
name="Jacob Savage"    (Bourne and Korn shells)
set name = "Jacob Savage"  (C shell)
```

(文件 datafile 中的行)

```
Jacob Savage:408-298-7732:934 La Barbara Dr. , San Jose, CA:02/27/78:500000
```

(nawk 命令行)

```
nawk -F: '$1 ~ /^'"$name"'/{print $2}'  datafile
```

(输出)

```
408-298-7732
```

尝试下面的例子：

1. 在插入任何 shell 变量之前，在命令行上测试你对 UNIX/Linux 命令的了解程度。

```
nawk -F: '$1 ~ /^Jacob Savage/{print $2}' filename
```

(输出)

```
408-298-7732
```



2. 仅插入 shell 变量，不改变其他任何东西，所有引号保持它们原来的样子。

```
nawk -F: '$1 ~ /^$name/{print $2}' datafile
```

从 awk 命令左边开始，将第一个引号保持原样，在$name 的 shell 美元符号前再放置一个单引号。现在第一个单引号匹配成功，这对引号中间的字符将不受 shell 干扰。而后面的变量不在引号内。接着在$name 中的字母 e 之后再加上一个单引号。于是又匹配成功一对单引号，这对单引号结束于 awk 右花括号之后。这一对引号中的所有字母也将不会被 shell 解释了。

```
nawk -F: '$1 ~ /^'$name'/{print $2}' datafile
```

3. 用一对双引号引用 shell 变量。这样该变量就能够被扩展了。但是，如果它包含有空白字符则该变量会被当作一个字符串处理。因此命令行要正确被解析则空白符必须被保护起来。

```
nawk -F: '$1 ~ /^'"$name"'/'{print $2}' datafile
```

计算引号的数量。单引号和双引号的数量都应该是一个偶数。

还有另外一个例子：

```
oldname="Ellie Main"
newname="Eleanor Quigley"
```

1. 确保命令能够工作。

```
nawk -F: '/^Ellie Main/{$1="Eleanor Quigley"; print $0}' datafile
```

2. 插入变量。

```
nawk -F: '/^$oldname/{$1="$newname"; print $0}' datafile
```

3. 进行引用游戏。从第一个单引号左侧开始，向行右侧移动直至到达变量$oldname，在这个美元符号前放置另外一个单引号。将另外一个单引号放置在该变量最后一字母之后。

现在移动到最右侧，在$newname 的美元符号前再放置一个单引号，另外一个单引号放置在$newname 最后一个字母之后。

```
nawk -F: '/^'$oldname'/{$1="'$newname'"; print $0}' datafile
```

4. 计算单引号的数量。如果单引号数为偶，则每个引号都有匹配的另外一个引号。如果不是偶数，那么你肯定是遗漏了某个单引号。

5. 用双引号引用所有的 shell 变量。双引号要恰好将 shell 变量引起来。

```
nawk -F: '/^'"$oldname"'/{$1="'"$newname"'"; print $0}' datafile
```

here 文档问题　shell 脚本中创建菜单的首选 here 文档通常是错误的来源。通常发生的问题是由用户定义的结束 here 文档的终止符引起的。终止符周围不能有空格。尽管

Bourne，bash 和 Korn 等 shell 在特定条件下允许使用 tab 键，但所有 shell 还是严格遵守这个规定。看下面的范例。

范例 15-11

```
#! /bin/ksh
   print "Do you want to see the menu?"
   read answer
   if [[ $answer = y ]]
   then
1      cat << EOF        <-- No space after user-defined terminator
       1) Steak and eggs
       2) Fruit and yogurt
       3) Pie and icecream .
2      EOF               <-- User defined terminator cannot
                             have spaces surrounding it
       print "Pick one "
       read choice
       case "$choice" in
           1) print "Cholesterol"
              ;;
           2) print "Dieter"
              ;;
           3) print "Sweet tooth"
              ;;
       esac
   else
       print "Later alligator!"
   fi
```

(输出)

```
file: line 6: here document 'EOF' unclosed
```

或

```
file: line 6: syntax error: unexpected end of file (bash)
```

说明

1. 这里是 here 文档的开始。cat 命令后接<<和一个用户定义的终止符，在这里是 EOF。终止符后面的行作为 cat 命令的输入，在屏幕上产生菜单选项。当到达第 2 行的终止符后输入结束。

2. 第 2 行的终止符必须恰好与第 1 行的终止符相匹配，否则 here 文档将不会结束。另外，最后的终止符周围不能有任何空格。有良好编程习惯的程序员往往会使用缩进来增强脚本的可读性，但这个例子中如果对第 2 行的 EOF 进行缩进将导致一个语法错误。解决此问题的方法是将用于终止的 EOF 移到页的最左侧，确保它周围没有任何空格。bash/ksh/sh 这 3 种 shell 允许使用另外一种方式。即在<<符号后加上一个长划线：cat <<-EOF。这样就可以使用 tab 来缩进第 2 行中用来结束输入的终止符。

文件测试错误 如果在一个脚本中使用外部文件，那么使用这些文件之前最好对它们的属性进行检查，如它们是否确实存在，文件是否可读或可写，文件是否有任何数据，是



否是符号链接等。各 shell 的文件测试十分相似，但对文件是否存在的测试有所不同。例如，C，TC 和 Bash 3 种 shell 使用-e 开关来检测文件是否存在，Korn shell 使用-a 开关，而 Bourne shell 使用-f 开关。除 TC shell 外，其他 shell 的文件测试开关不能与其他选项合在一起，比如像使用-rw 进行读写一样。作为代替，文件测试采用一个单独的文件测试开关加文件名的形式。C shell 测试文件是否可读、可写和可执行的一个例子为 if (–r filename && –w filename && –x filename)。TC shell 测试的例子为 if (–rwx filename)。

在 5 种 shell 中测试文件存在性

下面的错误信息是在脚本中执行文件测试之前产生的。文件 db 不存在。

```
grep: db: cannot open [No such file or directory]
```

下面的范例示意了 5 种 shell 中是如何解决这个问题的。

范例 15-12

```
(csh/tcsh)
set filedb = db
if ( ! -e $filedb ) then
  echo "$filedb does not exist"
  exit 1
endif

(sh)
filedb=db
if [ ! -f $filedb ]
then
  echo "$filedb does not exist"
  exit 1
fi

(ksh)
filedb=db
if [[ ! -a $filedb ]]
then
  print "$filedb does not exist"
  exit 1
fi

(bash)
filedb=db
if [[ ! -e $filedb ]]
then
  echo "$filedb does not exist"
  exit 1
fi
```

15.4.3 5 种 shell 中常见的错误信息

运行脚本时 shell 能够报告许多不同类型的语法错误。所有的 shell 都是通过向标准错

误输出发送一条消息来报告这些错误，尽管各 shell 的这些消息互不相同。例如，C shell 的错误报告十分详细，甚至会对发生错误的同一行报告不匹配的引用和未定义的变量两个错误。相比之下，Bourne shell 的错误信息很少并且不是很明确。调试 Bourne shell 脚本相当困难，这是因为直到整个脚本被分析完毕才报告错误，而且往往这些错误信息对定位错误毫无帮助。

因为各 shell 有自己的错误报告风格，所以表 15-3 至表 15-6 举例说明了最常见的语法错误、产生的原因、错误信息的含义以及简单的解决方式。

C/TC shell 常见的错误信息 表 15-3 列出了 C shell 常见的一些错误信息。因为 TC shell 与 C shell 非常相近，所以表中列出的内容对两种 shell 都适用。

表 15-3 C/TC shell 常见的错误信息

错误信息	原因	含义	解决方法
": Event not found.	echo "Wow!"	感叹号(历史字符)必须被转义。引号不能保护它	echo "Wow\!"
@: Badly formed number	set n = 5.6; @ n++; @ n = 3+4	算术运算只能针对整数，算术运算符前后必须有空格	set n = 5; @ n++; @ n = 3 + 4
@n++: Command not found	@n++	@符号后必须跟一个空格	@ n++
Ambiguous.	`date`	当某个命令的输出被赋给一个变量或将该输出内容作为某个字符串的一部分时，反引号用于命令替换。如果命令自成一行，则它不应该使用反引号。这种情况下 Tcsh 将报错"Fri:Command not found."	echo The date is `date` 或 set d = `date`
Bad :modifier in $ (f).	echo $cwd:f	:f 是一个无效的路径扩展名修饰符	echo $cwd:t 或 $cwd:h 等
Badly placed ()'s.	echo (abc)	圆括号用于启动新的子 shell。应该将整个命令放入( )中或将字符串用引号引起来	(echo abc) 或 echo " (abc) "
echo: No match.	echo How are you?	问号是一个用于文件名扩展的 shell 元字符。它代表文件名中的一个字符。shell 将匹配名为 you 后跟单个字符的文件。因为这里没有以此命名的文件，因此显示 No match 错误	echo "How are you? "或 echo 'How are you?'或 echo How are you\?
filex:File exists.	sort filex > temp	noclobber 变量被设置并且 temp 文件存在。noclobber 不允许改写一个已经存在的文件	sort filex > temp1(使用一个不同的文件名进行输出)或 unset noclobber 或 sort filex >! temp(改写 noclobber)



(续表)

错误信息	原因	含义	解决方法
fruit:Subscript out of range	echo $fruit[3]	fruit 数组不足 3 个元素	set fruit = (apples pears plums)
if: Empty if	if ($x > $y)	if 表达式不完整。缺少 then	if ($x > $y) then
if: Expression Syntax	if ($x = $y) then	if 相等操作符应为==	if ($x == $y) then
if: Expression Syntax	set name = "Joe Doe" if ($name = = "Joe Doe") then	= =左侧的变量 name 应该用双引号引用	if ("$name" == "Joe Doe") then
if: Expression Syntax	if (grep john filex) then	当对命令求值时，应该使用花括号而不是圆括号	if { grep john filex } then
Invalid null command	echo "hi" &> temp	重定向操作符是反向的。应该为>&	echo "hi" >&temp
Missing }.	if {grep john filex} then	花括号前后必须有空格	if { grep john filex } then
set:Syntax error	set name= "Tom"	等号两侧都应该有空格或都没有空格	set name = "Tom"或 set name="Tom"
set:Syntax error	set name.1 = "Tom"	变量名中的句点不是有效字符	set name1 = "Tom"
set:Syntax error	set file-text = "foo1"	变量名中的长划线是非法字符	set file_text = "foo1"
shift: No more words	shift fruit	shift 命令左移数组最左边的元素。导致这个错误的原因为 fruit 数组已经没有元素了。不能对一个空数组进行左移	set fruit = (apples pears plums)
then:then/endif not found	if ($x > &y) then statements statements	if 表达式不完整。缺少 endif	if ($x > &y) then statements endif
Too many ('s	if ($x = = $y && ($x != 3) then	表达式中的圆括号不对称。要么在最右侧增加一个，要么删除&&后的(	if ($x == $y && ($x != 3)) then 或 if ($x == $y && $x != 3) then
top:label not found.	goto top	goto 命令将查找程序中独立成行的标号 top。出错原因为或者标号不存在或者拼写有误	top: goto top
Undefined variable	echo $name	变量 name 未被设置	set name; set name = "John"; set name = ""
Unmatched".	echo She said, "Hello	双引号必须在一行内成对匹配	echo 'She said, "Hello"'
Unmatched'.	echo I don't care	单引号必须在一行内成对匹配	echo "I don't car"或 echo I don\'t care
while:Expression syntax.	while ($n=< 5)	操作符错误。正确的应该是<=	while ($n <= 5)

Bourne shell 常见的错误信息 表 15-4 列出了 Bourne shell 常见的错误信息

表 15-4 Bourne shell 常见的错误信息

错误信息	原因	含义	解决方法
./file: line 5: syntax error near unexpected token blue)	color="blue" case $color	case 命令缺少关键字 in	case $color in
[空行] shell 产生一空白行，没有错误信息	echo $name	变量不存在或为空	name="some value";
[ellie: not found	if [$USER = "ellie"] ; then	[之后至少要有一个空格	if [$USER = "ellie"] ; then
answer: not found	answer = "yes"	等号前后都不能有任何空格	answer="yes"
cannot shift	shift 2	内置命令 shift 用于将位置参量左移。如果位置参量的个数不足 2 个，则 shift 操作将会失败	set apples pears peaches; shift 2 (apples 和 pears 将被从列表中移走)
name: is read only	name="Tom"; readonly name; name="Dan"	变量设置成只读。它不能被重新定义或复位	name2="Dan" 或 exit shell
name: parameter not set	echo $name	设置了 set -u。以这个选项使用 set 命令，未定义的变量将被标记出来	name="some value";
name.1=Tom: not found	name.1="Tom"	变量名中句点不是有效字符	name1="Tom"
syntax error at line 7 'fi' unexpected	if [$USER = "ellie"] echo "hi" fi	表达式后面必须要有 then	if [$USER = "ellie"] then echo "hi" fi
syntax error: `{'	fun() {echo "hi"; }	函数 fun( )的定义中的花括号前后必须要有空格	fun() { echo "hi";; }
syntax error: 'done' unexpected	while [$n < 5] statements done	while 缺少关键字 do	while [$n -lt 5] do statements done
syntax error: 'fi' unexpected"	if [$USER = "ellie] then	then 应该另起一行或在前面加上一个分号	if [$USER = "ellie"] then 或 if [$USER = "ellie"]; then
test: argument expected	if [25 >= 24]; then	>=不是一个有效的测试操作符。应该为-ge	if [25 -ge 24]; then
test: unknown operator	if [grep $USER /etc/passwd]; then	grep 命令不应该被方括号或其他符号包围。此括号仅用于测试表达式。[是一个测试操作符	if grep $USER /etc/passwd; then



(续表)

错误信息	原因	含义	解决方法
test: unknown operator Doe	name="John Doe"; if [$name = Joe]	测试表达式中的变量 name 应该用双引号进行引用。除非被引用，否则测试操作符=的左侧最多只能有一个字符串	if ["$name" = Joe]
trap: bad trap	trap 'rm tmp*' 500	数字 500 是一个非法的信号。通常脚本中断使用信号 2 或 3，2 是 Ctrl-C，3 是 Ctrl-\。两个信号都会导致 trap 命令后面的程序被终止	trap 'rm tmp*' 2
unexpected EOF 或 unexpected end of file	echo "hi	脚本中的双引号不匹配。Bourne shell 将试着查找相匹配的引号直至文件尾。也可能不出现错误信息，但程序会输出意外的结果。如果 case 或循环命令不以它们各自的关键字 esac 和 done 结束，则 shell 将报告 unexpected EOF 错误	echo "hi"

Korn shell 常见的错误信息 表 15-5 列出了 Korn shell 常见的错误信息。

表 15-5 Korn shell 常见的错误信息

错误信息	原因	含义	解决方法
./file: line 5: syntax error near unexpected token blue)	case $color blue) ...	case 命令缺少关键字 in	case "$color" in blue) ...
.filename: line2:syntax error at line 6: ")" unexpected .	case $color in blue) echo "blue" red) echo "red" ;; esac	case 语句不是被 echo "blue" 后面的;；终止的	case $color in blue) echo "blue" ;; red) echo "red" ;; esac
[空白行]	echo $name 或 echo ${fruit[5]}	变量不存在或为空	name="some value"
file: line2: syntax error at line 6: 'done' unexpected	while (($n < 5)) statements done	while 缺少关键字 do	while ((n < 5)) do statements done
file: syntax error at line 3: ''' unmatched	print I don't care	脚本中的单引号不匹配。应该在它前面加上反斜线或用双引号进行引用	echo I don\'t care 或 echo "I don't care"

(续表)

错误信息	原　因	含　义	解决方法
file: syntax error at line 3: "" unmatched	print She said "Hello	脚本中的双引号不匹配	print She said "Hello"
ksh: [: Doe: unknown operator	name="John Doe" if [$name = Joe]; then	测试表达式中的变量 name 应该用双引号进行引用。除非被引用，否则测试操作符=的左侧最多只能有一个字符串。另一种可选的方式是使用复合测试操作符[[ ]]。使用复合测试操作符时，词不会被拆分	if ["$name" = Joe]; then 或 if [[$name = Joe]]; then
ksh: [ellie: not found	if [$USER = "ellie"]; then if [[$USER = "ellie"]]; then	[或[[之后至少要有一个空格	if [$USER = "ellie"]; then 或 if [[$USER = "ellie"]]; then
ksh: apples: bad number	set -A fruit apples pears peaches; shift fruit	内置命令 shift 不能对数组移位，它是用于对位置参量移位的	set apples pears peaches; shift
-ksh: file.txt=fool: not found	file.txt="fool"	变量名中不能含有句点	file_txt="fool"
ksh: filex: file already exists.	sort filex > temp	变量 noclobber 被设置且 temp 文件存在。noclobber 不允许改写已存在的文件	sort filex > temp1 (使用一个不同的文件进行输出) 或 set +o noclobber 或 sort filex > temp (改写 noclobber)
ksh: fred: unknown test operator	if [grep fred /etc/passwd]; then	grep 命令不应该被方括号或其他符号包围。此括号仅用于测试表达式。[是一个测试操作符	if grep fred /etc/passwd; then
ksh: name: not found	name = "Tom"	等号前后都不能有空格	name="Tom"
ksh: shift: bad number	shift 2	内置命令 shift 用于将位置参量左移。如果位置参量的数量不足 2 个，则 shift 操作将会失败	set apples pears peaches; shift 2 (apples and pears will be shifted from the list)
ksh: syntax error: '{echo' not expected	function fun {echo "hi"}	在函数 fun()的定义中花括号前后必须要有空格。另外必须使用一个分号来结束 function 语句	function fun { echo "hi"; }
ksh: syntax error: 'Doe' unexpected	if [[$name = John Doe]]	=号左侧的词是不会被拆分的，但是=号右侧的字符串必须被引用	if [[$name = "John Doe"]]



(续表)

错误信息	原因	含义	解决方法
ksh: syntax error: 'fi' unexpected"	if [$USER = "ellie] then	then 应该另起一行或在前面加上一个分号	if [$USER = "ellie"] then 或 if [$USER = "ellie"]; then
ksh: syntax error: 'then' unexpected	if ((n==5 && (n>3 || n<7))	包围第二个表达式的圆括号不匹配	if ((n==5 && (n>3 || n<7)))
ksh: trap: 500: bad trap ③	trap 'rm tmp*' 500	数字 500 是一个非法的信号。通常脚本中断使用信号 2 或 3，2 是 Ctrl-C，3 是 Ctrl-\。两个信号都会导致 trap 命令后面的程序被终止	trap 'rm tmp*' 2

常见的 bash 错误信息　表 15-6 列出了常见的 bash shell 错误信息。

表 15-6　bash shell 常见的错误信息

错误信息	原因	含义	解决方法
bash: syntax error: ' " unexpected EOF while looking for matching ' "	echo I don't care	脚本中的单引号不匹配	echo "I don't care"
bash: syntax error: ' " ' unexpected EOF while looking for matching ' "	print She said "Hello	脚本中的双引号不匹配	print She said "Hello"
空行，没有错误信息，没有输出	echo $name 或 echo ${fruit[5]}	变量不存在或变量为空	name="some value"或使用 set –u 捕获尚未传递的变量。消息 ksh: name: paramter not set 将被发往标准错误输出
bash: name: command not found	name = "Tom"	等号前后都不能有任何空格	name="Tom"
bash: 6.5: syntax error in expression (error token is ".5")	declare -i num; num=6.5	变量 num 只能赋予整数值	num=6
bash: [ellie: command not found	if [$USER = "ellie"]; then 或 if [[$USER = "ellie"]]; then	在[或[[之后要有一个空格	if [$USER = "ellie"]; then 或 if [[$USER = "ellie"]]; then

③　这个错误发生在公用版本的 Korn shell 上，但是 Korn shell 88 (Solaris)却不输出任何消息。

(续表)

错误信息	原因	含义	解决方法
bash: syntax error near unexpected token 'fi'	if [$USER = "ellie"] then	then 应该另起一行或跟在分号后面	if [$USER = "ellie"]; then 或 if [$USER = "ellie"] then
bash: syntax error in conditional expression	if [[$name = John Doe]]; then	=号左侧的词是不会被拆分的，但是=号右侧的字符串必须被引用	if [[$name == "John Doe"]]; then
[: fred: binary operator expected	if [grep fred /etc/passwd]; then	grep 命令不应该被方括号或其他符号包围。仅在测试表达式时才使用方括号。[是测试操作符	if grep fred /etc/passwd; then
./file: line 5: syntax error near unexpected token blue)	color="blue" case $color	case 命令缺少关键字 in	case $color in
.filename: line2: syntax error at line 6: ")" unexpected.	case $color in blue) echo "blue" red) echo "red" ;; esac	case 语句不是被 echo "blue"后面的;; 终止的	case $color in blue) echo "blue" ;; red) echo "red" ;; esac
bash: shift: bad non-numeric arg 'fruit'	declare -a fruit=(apples pears peaches); shift fruit	内置命令 shfit 不能对数组移位。它只能用来对位置参量进行移位	set apples pears peaches; shift
[: too many arguments	name="John Doe"; if [$name = Joe]	测试表达式中的变量 name 应该用双引号进行引用。除非被引用，否则测试操作符=的左侧最多只能有一个字符串。另一种可选的方式是使用复合测试操作符[[ ]]。使用复合测试操作符时，词不会被拆分	if ["$name" = Joe] 或 if [[$name == Joe]]
bash: syntax error near unexpected token '{echo'	function fun {echo "hi"}	在函数 fun()的定义中花括号前后必须要有空格。另外必须使用一个分号来结束 function 语句	function fun { echo "hi"; }



(续表)

错误信息	原　因	含　义	解决方法
bash: filex: cannot overwrite existing file	sort filex > temp	变量 noclobber 被设置且 temp 文件存在。noclobber 不允许改写已存在的文件	sort filex > temp1 (使用一个不同的文件来进行输出)或 set +o noclobber or sort filex >| temp (改写 noclobber)
bash:trap: 500: not a signal specification	trap 'rm tmp*' 500	数字 500 是一个非法的信号。通常脚本中断使用信号 2 或 3，2 是 Ctrl-C，3 是 Ctrl-\。两个信号都会导致 trap 命令后面的程序被终止	trap 'rm tmp*' 2

15.4.4 逻辑错误与健壮性

逻辑错误通常难以发现，这是因为逻辑错误并不一定会导致错误信息，而是导致程序行为异常。这样的错误通常可能是误用关系、相等或逻辑操作符，在一组嵌套条件语句中出现分支错误，或进入了一个无限循环。健壮性是指通过执行充分的错误检查就能够定位的错误，如检查用户错误的输入，参数不足或变量中的空值。

逻辑操作符错误　范例 15-13 示意了一个逻辑操作符错误并给出了一种可能的解决方法。

范例 15-13

```
#!/bin/csh
1  echo -n "Enter -n your grade: "
   set grade = $<
2  if ( $grade < 0 && $grade > 100 ) then
3     echo Illegal grade.   # This line will never be executed
      exit 1
   endif
4  echo "This line will always be executed."

(输出)
Enter your grade: -44
This line will always be executed.

Enter your grade: 234
This line will always be executed.
Enter your grade

------------------Possible Fix----------------

if ( $grade < 0 || $grade > 100 ) then
```

说明

1. 要求用户进行输入。输入被赋给变量 grade。

2. 逻辑操作符与(&&)要求两个表达式都为真才能进入第 3 行的 if 块。如果两个表达式均为真，则用户应该输入一个既小于 0 同时又大于 100 的数，而这是不可能的。因此在这里应该使用逻辑操作符或(| |)。这样只要求一个条件为真即可。

3. 这一行永远也不会被执行。

4. 因为第 2 行永不为真，所以这条语句总是执行。

关系操作符错误 范例 15-14 示意了一个关系操作符错误并给出了一种可能的解决方法。

范例 15-14

```
#!/bin/csh
    echo -n "Please enter your age "
    set age = $<

1   if ( $age > 12 && $age < 19 ) then
        echo A teenager            # What if oyou enter 20?
2   else if ( $age > 20 && $age < 30 ) then
        echo A young adult
3    else if ( $age >= 30 ) then    # What if the user enters 125?
        echo Moving on in years
    else
4       echo "Invalid input"
    endif

(输出)
Please enter your age  20
Invalid input

Please enter your age  125
Moving on in years

------------------Possible Fix----------------

if ( $age > 12 && $age <= 19 ) then
    echo A teenager
else if ( $age >= 20 && $age < 30 ) then
     echo A young adult
else if ( $age >= 30  && $age < 90 ) then
   echo Moving on in years
else if ( $age <=12 ) then
   echo still a kid
else
   echo "Invalid input"
endif
```

说明

1. 该表达式测试年龄是否在 13~18 之间。要对 13~20 之间的年龄进行测试，则右侧



表达式有两种改变方式，或者改为($age <= 19)，或者改为($age < 20)。

2. 如果年龄为 19，则程序总是转向第 3 行。该表达式用来测试 21~29 之间的年龄。我们需要让该表达式包含 20。如果用户输入 19 或 20，则程序打印“Invalid input”。

3. 该表达式测试大于 29 的年龄。这里没有设置上限。除非用户可以无限老，否则表达式应该包含上界。

4. 无效输入为 19、20 以及任何小于 13 的数。

分支错误　范例 15-15 示意了一个分支错误并给出了一种可能的解决方法。

范例 15-15

```
1   set ch = "c"
2   if ( $ch == "a" ) then
3       echo $ch
4       if ( $ch == "b" ) then        # This "if" is never evaluated
            echo $ch
5   else
6       echo $ch
7   endif
8   endif

(输出)
<没有输出>

------------------Possible Fix----------------

set ch = "c"
if ( $ch == "a" ) then
   echo $ch
else if ( $ch == "b" ) then
        echo $ch
else
    echo $ch
endif
```

说明

1. 变量 ch 被赋值为字母 c。

2. $ch 的值为 a，第 3~5 行被执行。

3. 如果$ch 的值为 a，则$ch 的值将被打印，程序继续执行第 4 行嵌套的 if。除非$ch 为 a，否则该语句决不会执行。

4. 如果$ch 为 a，自然它不会为 b，因此第 6 行不会执行。

5. else 应该以缩进方式置入到内部的 if 中。else 与第 4 行最里面的 if 相匹配。

6. 当且仅当$ch 为 a 时，本行被执行。

7. 该 endif 与第 4 行最里面的 if 相匹配。

8. 该 endif 与第 2 行外面的 if 相匹配。

if 条件句使用的退出状态值　如果 if 命令要检测某个命令执行成功或失败，则检查该命令的退出状态值。如果退出状态为 0 则该命令执行成功，如果退出状态非 0 则该命令执

行失败。如果不能确定命令返回的退出状态值，则应该在使用之前对它进行检查。否则你的程序可能不能按预期执行。考虑下面这个例子。awk，sed 和 grep 命令均可以使用正则表达式来搜索模式，但 grep 是 3 个命令中唯一的在不能搜索到相应模式时报告非 0 退出状态的命令。无论是否找到搜索模式，sed 和 awk 程序均返回 0。因此，这两个程序都不能用在 if 条件句中，因为该条件可能永远为真。

范例 15-16

```
#!/bin/sh
1  name="Fred Fardbuckle"
2  if grep "$name" db > /dev/null 2>&1
   then
3     echo Found $name
   else
4     echo "Didn't find $name"      # Fred is not in the db file
   fi
5  if awk "/$name/" db > /dev/null 2>&1
   then
6     echo Found $name
   else
      echo "Didn't find $name"
   fi
7  if sed -n "/$name/p" db > /dev/null 2>&1
   then
8     echo Found $name
   else
      echo "Didn't find $name"
   fi
```

（输出）

```
4 grep: Didn't find Fred Fardbuckle
6 awk: Found Fred Fardbuckle
8 sed: Found Fred Fardbuckle

------------------------Possible Fix--------------------------

Check the exit status of the command before using it.

     awk "/$name/" db
     echo $? (bash, sh, ksh)
     echo $status (tcsh, csh, bash)
```

说明

在这个范例中我们可以看到，除非命令语句有误，否则 awk、nawk 和 gawk 总是返回退出状态 0。grep 在搜索成功时返回退出状态 0，如果不能在文件中找到模式或文件根本不存在则返回一个非 0 整数值。

健壮性不足　如果程序能够合理地处置非法输入以及其他意外情况，则称该程序是健壮的。例如，假设程序需要一个数值型数据，则它必须能够检测出当前数据的类型，如果



是错误的类型，则应该打印错误信息并忽略该输入。另外一个健壮性的例子：假设脚本需要从外部文件中提取数据，但这个外部文件不存在或不允许读。一个健壮的程序应该在读取文件之前先对文件进行测试看文件是否存在。

范例 15-17 示意了如何检查空输入。

范例 15-17

```
#!/bin/csh
# Program should check for null input -- T and TC shells

1   echo -n "Enter your name: "
       set name = $<        # If user enters nothing, program will hang
2   if { grep "$name" db >& /dev/null } then
       echo Found name
    endif

(输出)
Enter your name:  Ellie
Found name

Enter your name:
    <程序打印出文件 Found name 中的每一行>

--------------------Possible Fix------------------------------------

   echo -n "Enter your name: "
   set name = $<    # If user enters nothing, program will hang
3   while ( $name == "" )
     echo "Error: you did not enter anything."
     echo -n "Please enter your name"
     set name = $<
   end
     <程序在此继续>
```

说明

1. 要求用户进行输入。如果用户直接按下回车键，则变量 name 将被设置为空(null)。

2. 程序第一次运行时，用户键入一些信息，然后 grep 在文件中搜索该模式。下一次，用户直接按下回车键，则变量被设置为空，导致 grep 程序搜索空值。所有的行均被打印。因为错误和输出均被发送到/dev/null，因此不能明确 grep 打印整个文件内容的原因。

3. 循环测试变量 name 中的空值。循环将持续进行直到用户输入一些信息而不再是直接按下回车键。对 ksh、bash 和 sh，要使用正确的 while 循环语法(例如，while [$name = " "] 或 while [[$name = " "]])。参见语法中使用的特殊字符。

范例 15-18 和范例 15-19 示意了如何检查参数不足。

范例 15-18

```
#!/bin/sh
# Script: renames a file -- Bourne shell
```

```
1  if [ $# -lt 2 ]      # Argument checking
2  then
      echo "Usage: $0 file1 file2 " 1>&2
      exit 1
   fi
3  if [ -f $1 ]         # Chneck for file existence
   then
      mv $1 $2          # Rename file1
      echo $1 has been renamed to $2
  else
      echo "$1 doesn't exist"
      exit 2
   fi
```

（输出）

```
$ ./rename file1
Usage: mytest file1 file2
```

说明

1. 如果位置参量(参数)的个数小于 2，则继续下一步。
2. 错误信息被发往标准错误输出，程序退出。退出值为 1，暗示程序运行时出现问题。
3. 当从命令行传来的参数不足时程序继续运行。检查语法的正确性以将该程序移植到 ksh 或 bash 上。

范例 15-19

```
   #!/bin/csh
   # Script: renames a file -- C/TC shells

1  if ( $#argv < 2 ) then       # Argument checking
2     echo "Usage: $0 file1 file2 "
3     exit 1
   endif
   if ( -e $1 ) then            # Check for file existence
      mv $1 $2                  # Rename file1
      echo $1 has been renamed to $2
   else
      echo "$1 doesn't exist"
      exit 2
   endif
```

（输出）

```
% ./rename file1
Usage: mytest file1 file2
```

范例 15-20 和范例 15-21 示意了如何检查输入是否为数值

范例 15-20

（脚本）

```
   $ cat trap.err
```



```
    #!/bin/ksh
    # This trap checks for any command that exits with a nonzero
    # status and then prints the message -- Korn shell

1   trap 'print "You gave me a non-integer. Try again. "' ERR
2   typeset -i number        # Assignment to number must be integer
3   while true
    do
4       print -n "Enter an integer. "
5       read -r number 2> /dev/null
6       if (( $? == 0 ))     # Was in integer read in?
        then                 # Was the exit status zero?
7           break
        fi
    done
8   trap -ERR                # Unset pseudo trap for ERR
9   if grep ZOMBIE /etc/passwd > /dev/null 2>&1
    then
        :
    else
10      print "\$n is $n. So long"
    fi
```

(命令行与输出)

```
    $ trap.err
4   Enter an integer. hello
1   You gave me a non-integer. Try again.
4   Enter an integer. good-bye
1   You gave me a non-integer. Try again.
4   Enter an integer. \\\
1   You gave me a non-integer. Try again.
4   Enter an integer. 5
10  $n is 5. So long.

    $ trap.err
4   Enter an integer. 4.5
$n is 4. So long.
```

范例 15-21

(脚本)

```
    #!/bin/bash
1   # Scriptname: wholenum
    # Purpose:The expr command tests that the user enters an integer -- Bash shell

    echo "Enter an integer."
    read number
2   if expr "$number" + 0 >& /dev/null
    then
3       :
    else
```

```
4       echo "You did not enter an integer value."
        exit 1
fi
```

15.5 使用 shell 选项与 set 命令进行跟踪

15.5.1 调试 Bourne shell 脚本

Bourne shell 调试选项 通过对 sh 命令使用-n 选项，可以在不用执行任何命令的情况下检查脚本的语法。如果脚本中有语法错误，shell 将报告这些错误，如果没有错误，则什么也不显示。

调试脚本最常用的方式是用带-x 选项的 set 命令(set -x)或使用以-x 选项为参数的 sh 命令(sh -x)，后跟脚本名。表 15-7 列出了调试选项。这些选项可以跟踪脚本的执行。执行替换之后，脚本中的命令先被显示出来，然后再执行。脚本中的每一行之前都会先带上一个加号(+)。

通过开启 verbose 选项(set -v)或以-v 选项调用 Bourne shell，脚本的每一行都被显示，与键入时的完全相同，然后脚本被执行。

表 15-7 Bourne shell 调试选项

命 令	选 项	含 义
sh –x scriptname	Echo 选项	在变量替换之后，执行之前显示脚本的每一行
sh –v scriptname	Verbose 选项	执行之前显示脚本的每一行，与键入脚本中的一样
sh –n scriptname	Noexec 选项	解释但不执行命令
set –u	未绑定变量	尚未设置的标志变量
set –x	开启 echo	跟踪脚本执行
set +x	关闭 echo	关闭跟踪

set 命令可以在调试脚本的某一部分而非整个程序时，开启或关闭回显。

格式

```
set -x     # turns echoing on
   <程序语句>
set +x     # turns echoing off
```

范例 15-22

(脚本)

```
$ cat todebug
   #!/bin/sh
1  # Scriptname: todebug
   name="Joe Blow"
   if [ "$name" = "Joe Blow" ]
```



```
then
    echo "Hi $name"
fi

num=1
while [ $num -lt 5 ]
do
    num=`expr $num + 1`
done
echo The grand total is $num
```

(输出)

```
2   $ sh -x todebug
    + name=Joe Blow
    + [ Joe Blow =  Joe Blow ]
    + echo Hi Joe Blow

    Hi Joe Blow
    num=1
    + [ 1 -lt 5 ]
    + expr 1 + 1
    num=2
    + [ 2 -lt 5 ]
    + expr 2 + 1
    num=3
    + [ 3 -lt 5 ]
    + expr 3 + 1
    num=4
    + [ 4 -lt 5 ]
    + expr 4 + 1
    num=5
    + [ 5 -lt 5 ]
    + echo The grand total is 5
    The grand total is 5
```

说明

1. 脚本名为 todebug，可以在-x 开关开启的情况下查看脚本运行。循环的每一次遍历都被显示出来，变量被设置或其值发生变化时，打印它们的值。

2. sh 命令以-x 选项为参数启动 Bourne shell。回显被打开。脚本的每一行被预先加上一个加号(+)并显示在屏幕上。这些行显示之前会执行变量替换。命令执行的结果在脚本被显示之后显示。

15.5.2 调试 C/TC shell 脚本

C shell 脚本通常因为一些简单的语法或逻辑错误而运行失败。这里提供了 csh 命令的选项以帮助您调试程序。参见表 15-8。

表 15-8 echo(-x)与 verbose(-v)

作为 csh 的选项(tcsh 同样适用)	
csh –x scriptname	在变量替换之后，执行之前显示脚本的每一行
csh –v scriptname	执行之前显示脚本的每一行，与键入脚本中的一样
csh –n scriptname	解释但不执行命令
作为 set 命令参数	
set echo	在变量替换之后，执行之前显示脚本的每一行
set verbose	执行之前显示脚本的每一行，与键入脚本中的一样
作为脚本的首行	
#!/bin/csh -xv	同时打开 echo 和 verbose 选项。这两个选项可以单独被调用或与其他 csh/tcsh 调用参数组合起来被调用

范例 15-23

(-v 和-x 选项)

```
1   % cat practice
    #!/bin/csh
    echo Hello $LOGNAME
    echo The date is `date`
    echo Your home shell is $SHELL
    echo Good-bye $LOGNAME

2   % csh -v practice
    echo Hello $LOGNAME
    Hello ellie
    echo The date is `date`
    The date is Sun May 23 12:24:07 PDT  2004
    echo Your login shell is $SHELL
    Your login shell is /bin/csh
    echo Good-bye $LOGNAME
    Good-bye ellie

3   % csh -x practice
    echo Hello ellie
    Hello ellie
    echo The date is `date`
    date
    The date is Sun May 23 12:24:15 PDT  2004
    echo Your login shell is /bin/csh
    Your login shell is /bin/csh
    echo Good-bye ellie
    Good-bye ellie
```

说明

1. 显示了 C shell 脚本内容。其中包含了具有变量替换和命令替换的行，这样可以看



清楚 echo 与 verbose 的差别。

2. csh 命令的-v 选项开启了 verbose 特性。脚本中的每一行以其键入脚本中的样子被显示，然后被执行。

3. csh 命令的-x 选项开启回显特性。脚本的每一行在执行完变量替换和命令替换后被显示，然后再被执行。因为这个特性可以检查命令替换和变量替换的实际效果，因此它比 verbose 选项要常用。

范例 15-24

```
(Echo)
1   % cat practice
    #!/bin/csh
    echo Hello $LOGNAME
    echo The date is `date`
    set echo
    echo Your home shell is $SHELL
    unset echo
    echo Good-bye $LOGNAME
    % chmod +x practice

2   % practice
    Hello ellie
    The date is Sun May 26 12:25:16 PDT 2004
--> echo Your login shell is /bin/csh
--> Your login shell is /bin/csh
--> unset echo
     Good-bye ellie
```

说明

1. 脚本中先设置接着又复位了 echo 选项。这样做可以调试脚本运行中出现瓶颈的部分，而不用回显整个脚本的所有行。

2. -->标志着回显被打开的位置。变量替换和命令替换后，所有行都被打印，然后被执行。

范例 15-25

```
(Verbose)
1   % cat practice
    #!/bin/csh
    echo Hello $LOGNAME
    echo The date is `date`
    set verbose
    echo Your home shell is $SHELL
    unset verbose
    echo Good-bye $LOGNAME

2   % practice
    Hello ellie
    The date is Sun May 23 12:30:09 PDT  2001
```

```
--> echo Your login shell is $SHELL
--> Your login shell is /bin/csh
--> unset verbose
   Good-bye ellie
```

说明

1. 脚本中先设置接着又复位了 verbose 选项。

2. -->标志着 verbose 被打开的位置。各行按照它们在脚本中被键入的样子显示，然后被执行。

15.5.3 调试 Korn shell 脚本

通过打开 noexec 选项或对 ksh 命令使用-n 参数，可以在不用执行任何命令的情况下检查脚本的语法错误。如果脚本中的确有语法错误，则 shell 将报告这些错误。如果没有任何错误，则什么也不显示。

最常用的调试脚本的方法是打开 xtrace 选项或对 ksh 命令使用-x 选项。这些选项可以跟踪脚本的执行过程。脚本中的每条命令在执行变量替换后总是先被显示出来，然后再执行。当显示脚本中的行时，总是先显示一个值为 PS4 的提示符，也就是一个加号(+)。PS4 提示符可以被更换。

通过打开 verbose 选项或以-v 为选项调用 Korn shell(ksh –v scriptname)，脚本中的所有行都将以它们被键入脚本时的样子被显示，然后再被执行。表 15-9 是一些调试命令。

表 15-9 Korn shell 调试命令和选项

命　令	功能/工作原理
export PS4='$LINENO '	PS4 提示符默认为+。可以重设这个提示符。在本例中，会为每行打印一个行号
ksh –n scriptname	以 noexec 选项调用 ksh。解释但并不执行命令
ksh -u scriptname	检测未设置的变量。而扩展未设置变量将显示错误
ksh –v scriptname	以 verbose 为选项调用 ksh。在执行脚本前，先按照脚本中的各行被键入时的样子显示每一行
ksh –x scriptname	以 echo 为选项调用 ksh。在变量替换之后，脚本执行之前显示脚本的每一行
set +x	关闭 echo。关闭跟踪
set –x 或 set –o xtrace	打开 echo 选项。跟踪脚本中执行过程
trap 'print $LINENO ' DEBUG	对每个脚本命令执行 trap 动作。参见 trap 的格式。打印脚本中每一行的$LINENO 值
trap 'print Bad input' ERR	如果返回了非 0 的退出状态值，则执行 trap
trap 'print Exiting from $0' EXIT	当脚本或函数退出时显示消息
typeset –ft	打开跟踪。跟踪函数执行

范例 15-26

（脚本）

```
    #!/bin/ksh
    # Scriptname: todebug
1   name="Joe Blow"
2   if [[ $name = [Jj]* ]] then
        print Hi $name
    fi
    num=1
3   while (( num < 5 ))
    do
4          (( num=num+1 ))
    done
5   print The grand total is $num
```

（命令行与输出）

```
1   $ ksh -x todebug
2   + name=Joe Blow
    + [[ Joe Blow = [Jj]* ]]
    + print Hi Joe Blow
    Hi Joe Blow
    + num=1                 # The + is the PS4 prompt
    + let num < 5
    + let num=num+1
    + let num < 5
    + let num=num+1
    + let num < 5
    + let num=num+1
    + let num < 5
    + let num=num+1
    + let num < 5
    + print The grand total is 5
    The grand total is 5
```

说明

1. 以-x为选项调用 Korn shell。打开了回显。脚本的每一行被显示在屏幕上，后面跟着该行的执行结果。执行了变量替换。另外，还可以直接在脚本中使用-x选项而不用在命令行使用(也就是，#!/bin/ksh -x)。
2. 每行前有一个加号(+)，也就是 PS4 提示符。
3. 进入了 while 循环，它将循环 4 次。
4. num 的值加 1。
5. while 循环退出后，打印该行。

范例 15-27

（脚本）

```
    #!/bin/ksh
    # Scriptname: todebug2
1   trap 'print "num=$num on line $LINENO"' DEBUG
```

```
num=1
while (( num < 5 ))
do
    (( num=num+1 ))
done
print The grand total is $num
```

(输出)

```
2  num=1 on line 3
   num=1 on line 4
   num=2 on line 6
   num=2 on line 4
   num=3 on line 6
   num=3 on line 4
   num=4 on line 6
   num=4 on line 4
   num=5 on line 6
   num=5 on line 4
   The grand total is 5
   num=5 on line 8
   num=5 on line 8
```

说明

1. LINENO 是一个特殊的 Korn shell 变量，它保存脚本当前行的行号。trap 命令使用的 DEBUG 信号导致每执行脚本中的一条命令就执行一次单引号中的字符串。

2. while 循环执行的过程中，变量 num 的值和脚本各行的行号被显示出来。

15.5.4 调试 bash 脚本

通过对 bash 命令使用-n 选项，可以在不用执行任何命令的情况下检查脚本的语法。如果脚本中有语法错误，shell 将报告这些错误，如果没有错误，则什么也不显示。

调试脚本最常用的方式是用带-x 选项的 set 命令或使用以-x 选项为参数的 bash 命令，后跟脚本名。表 15-10 列出了调试选项。这些选项可以跟踪脚本的执行。执行替换之后，脚本中的命令先被显示出来，然后再执行。脚本中的每一行显示之前，都会先显示一个加号(+)。

通过打开 verbose 选项或以-v 为选项调用 shell(bash –v scriptname)，脚本中的所有行都将以它们被键入脚本时的样子被显示，然后再被执行。

表 15-10 bash 调试选项

命 令	选 项	含 义
bash –x scriptname	Echo 选项	在变量替换之后，执行之前显示脚本的每一行
bash –v scriptname	Verbose 选项	执行之前显示脚本的每一行，与键入脚本中的一样
bash –n scriptname	Noexec 选项	解释但不执行命令
set –x	打开 echo	跟踪脚本执行
set +x	关闭 echo	关闭跟踪

bash 调用选项 当使用 bash 命令启动 shell 时，可以用选项调整 shell 的行为。有两种类型的选项：单字符选项和多字符选项。单字符选项由一个长划线加上单个字符构成。多字符选项由两个长划线加上任意多个的字符构成。多字符选项必须位于单字符选项之前。一个交互式登录 shell 通常以-i(启动交互式 shell),-s(从标准输入读取)和-m(打开作业控制)启动。参见表 15-11。

表 15-11 辅助调试的 bash 附加选项

选 项	含 义
-i	shell 处于交互模式。忽略 TERM、QUIT 和 INTERRUPT
-r	启动一个受限的 shell
--	标志着选项结束，禁止进一步的选项处理。--或-之后的参数被当作文件名及其参数
--help	显示有关内置命令信息并退出
--noprofile	启动时，bash 不读取初始化文件/etc/profile、~/.bash_profile、~/.bash_login 或~/.profile
--norc	对交互式 shell，bash 不读取~/.bashrc 文件。如果以 sh 运行 shell 则它默认为打开
--posix	改变 bash 的行为以符合 POSIX 1003.2 标准，如果不加上该选项，则不会遵循该标准
--quiet	shell 启动时不显示信息，这是默认的。
--rcfile file	如果 bash 是交互式的，则使用该初始化文件以替代~/.bashrc
--restricted	启动一个受限的 shell
--verbose	打开 verbose。与-v 相同
--version	显示该 bash shell 的版本信息并退出

set 命令与选项 set 命令可用来打开或关闭 shell 选项，也可用于处理命令行参数。要打开某个选项，可用长划线(-)加上该选项。要关闭某个选项，则用加号(+)加上该选项。

表 15-12 内置命令 set 的选项

选 项 名	快 捷 开 关	作 用
allexport	-a	该选项被设置后，将自动标记新创建或修改的导出变量用于输出直至选项复位
braceexpand	-B	允许括号扩展，它是默认的设置[④]
emacs		用内置编辑器 emacs 进行命令行编辑，它是默认设置
errexit	-e	如果命令返回非 0 的退出状态(失败)，则退出。读取初始化文件时该选项未设置
histexpand	-H	当执行历史替换时，开启!和!!选项，它是默认设置[④]
history		开启命令行历史，它是默认设置[④]
ignoreeof		禁用 EOF(Ctrl+D 组合键)退出 shell。必须键入 exit。与设置 shell 变量 IGNOREEOF=10 效果相同

④ 选项仅适用于 bash 版本 2.x。

(续表)

选 项 名	快 捷 开 关	作 用
keyword	-k	将关键字参数放入命令环境[④]
interactive-comments		对交互式 shell，在行首以一个#注释该行其余的部分
monitor	-m	允许作业控制
noclobber	-C	使用重定向时防止文件被改写
noexec	-n	读命令，但并不执行。用于检查脚本中的语法。交互地运行时不打开该选项
noglob	-d	禁用路径名扩展(也就是关闭通配符)
notify	-b	当后台作业结束时通知用户
nounset	-u	扩展未设置的变量时显示一个错误
onecmd	-t	读取并执行一个命令后退出[⑤]
physical	-P	如果被设置，则键入 cd 或 pwd 时，后面不能跟符号链接。而使用物理目录代替
posix		如果默认操作不符合 POSIX 标准则 shell 行为被改变
privileged	-p	设置后，shell 不读取.profile 或 ENV 文件，且不从环境中继承 shell 函数。setuid 脚本会自动设置该选项
verbose	-v	打开 verbose 模式用于调试
vi		使用内置编辑器 vi 进行命令行编辑
xtrace	-x	打开 echo 模式用于调试

shopt 命令与选项 shopt(bash 2.x)命令也可以用于打开和关闭 shell 选项。

表 15-13 shopt 命令选项

选 项	含 义
cdable_vars	如果内置命令 cd 的参数不是目录，则视它为一个变量名，并试着切换到其值所代表的目录
cdspell	更正 cd 命令中目录名的一些小的拼写错误。检查的错误类型有字符顺序颠倒、遗漏字符以及字符重复。如果找到一个更正方式，则打印更正后的路径，命令继续执行。仅用于交互式 shell
checkhash	bash 在执行命令前首先检查哈希表中是否存在该命令。如果不存在此哈希命令，则执行标准的路径搜索
checkwinsize	bash 在执行每条命令后检查窗口大小。如果有必要，则更新 LINES 和 COLUMNS 的值
cmdhist	bash 尝试在同一个历史目录中保存一个多行命令的所有行。这使得多行命令的再次编辑更加简单
dotglob	bash 包含文件名扩展后以点(.)开头的文件名
execfail	如果一个非交互的 shell 不能执行作为参数传给内置命令 exec 的文件，它将不会退出。交互式 shell 在 exec 失败时并不退出



(续表)

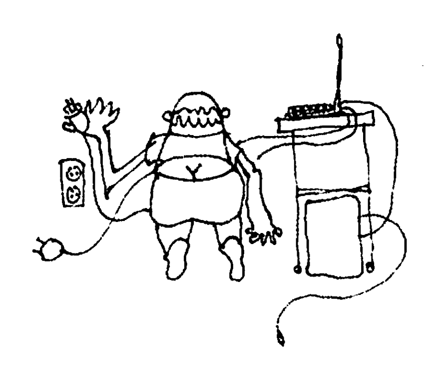
选　项	含　义
expand_aliases	扩展别名。默认为打开
extglob	开启扩展的模式匹配特性(使用来自 Korn shell 的正则表达式元字符进行文件名扩展)
histappend	shell 退出时，历史列表被追加到变量 HISTFILE 所代表的文件中，而并不改写该文件
histreedit	如果使用 readline，则用户有机会重编辑失败的历史替换
histverify	设置后，若同时又使用了 readline，则历史替换的结果并不立即传递给 shell 解析器。作为代替，结果行被载入到 readline 编辑器缓冲，从而允许进一步的修改
hostcomplete	设置后，若同时又使用了 readline，则当某个词包含一个可以被补全的@时，bash 将尝试执行主机名补全。该选项默认为打开
huponexit	设置后，在交互式 shell 退出时，bash 将向所有的作业发送 SIGHUP 信号(挂起信号)
interactive_comments	在交互式 shell 中，允许通过在某个词前加上#，从而注释该词以及该行后续所有字符。该选项为默认设置
lithist	如果设置，则同时也设置了 cmdhist 选项，将使用换行符而不是分号分隔符在历史中保存多行命令
mailwarn	设置后，bash 在检查邮件时，如果所检查的文件是上次已经访问过的，则显示信息 The mail in mailfile has been read
nocaseglob	设置后，则 bash 在执行文件名扩展时，将不考虑大小写来匹配文件名
nullglob	设置后，则 bash 将不能匹配到任何文件的文件名模式扩展为空字符串，而不是扩展为它们自身
promptvars	设置后，提示字符串在被扩展之后再进行变量和参数扩展。该选项为默认设置
restricted_shell	如果 shell 以受限模式启动，shell 设置该选项。其值不能被改变。当执行启动文件时，该选项没有被复位，从而允许启动文件查明该 shell 是否受限
shift_verbose	设置后，则内置命令 shift 在位移数超过位置参量个数时显示一条错误信息
sourcepath	设置后，则内置命令 source 使用 PATH 值查找包含参数文件的目录。该选项为默认设置
source	与点(.)命令同义

15.6 小结

现在我们已经介绍了主流的 UNIX 和 Linux shell，您可以读写并维护脚本了。记住，要把大量时间花在对脚本的调试上。因为常常有这样的情况，脚本已经可以正常运行了，但我们希望将它变得更好。于是，重返编辑器，对脚本进行一些修改，然后重新运行程序，很不幸，程序被破坏了！学习本章之后，您能够将这种情况减到最少。如果您是一个系统管理员，希望了解更多 shell 与系统交互的相关知识，下一章的内容会对此有所帮助，它主要介绍系统管理员使用 shell 的典型方式。

chapter

16



系统管理员与 shell

16.1 简介

系统管理员要比普通用户拥有更多的经验以担任特定的工作，例如修改引导脚本、增加用户、修复安装软件、监视进程运行、加载文件系统、备份等。因为很多的系统任务可以通过 shell 脚本自动进行，所以系统管理员需要掌握 shell 编程的相关知识，从而可以阅读并修改已经存在的脚本，一旦有必要，还可以创建新的脚本。本章不是关于系统管理的指南，它实际上是介绍如何使用 UNIX/Linux shell 进行系统管理。它所涵盖的内容包括以根用户(root)身份运行 shell 脚本、系统启动脚本、shell 初始化脚本以及如何编写可移植的 shell 脚本。我们将提供基于特定 UNIX/Linux 版本的可在各种系统上运行的范例。如果需要了解更多细节，请查阅所使用版本的文档资料。

如果您熟悉系统管理，将会发现本章的内容有助于您查缺补漏。如果您是系统管理方面的新手，则将会对那些非特权用户不能使用的 shell 相关内容有所领悟。

16.2 超级用户

若一个 UNIX 或 Linux 的新手忘记了自己的口令，并向同事询问该怎么办。典型的回答是“你自己无法解决这个问题，除非你是根用户。去找超级用户吧！”。在详细了解超级用户(也称为根用户)如何运行脚本的细节之前，首先要知道超级用户这个词是什么意思。UNIX/Linux 系统有两种类型的账户：普通用户和超级用户。普通用户仅可以访问属于他们自己的文件和进程，或者是被赋予了特定权限的，如组权限的文件和进程。而超级用户可以访问系统上所有的文件和进程。超级用户不需要获得任何许可就能够对属于其他用户的文件进行修改、复制、移动、检查和修改权限、删除等操作。能够杀死任意的进程而无需是进程的拥有者。他们几乎无所不能，不受任何限制。最常用的超级用户账户称为根用户，



很多机器还有其他的超级用户账户。可以通过运行 id 命令或者查看提示符来识别超级用户账户。如果 id 命令的结果显示用户标识符(uid)为 0，或者 shell 提示符是一个#号，那么这个账户就属于一个超级用户。术语“超级用户”和“根用户”是等价的。

范例 16-1

```
1  # id
   uid=0(root) gid=0(root) groups=0(root)
2  # ls -l /tmp
   total 1
   drwxr-xr-x    2  root     root       72   Feb 10 23:29 .
   drwxr-xr-x   26  root     root      680   Feb 10 23:28 ..
   -r--------    1  ellie    users       0   Feb 10 23:29 myfile
3  # cat myfile
   This is my file.
```

说明

1. 首先，注意提示符是一个#号。超级用户账户通常使用这样的提示符。当运行 id 命令时，输出显示用户标识号(用户标识号是您的账户号，即位于所在机器上的/etc/passwd 文件的第 3 个字段)为 0，说明这是一个超级用户账户。输出还显示登录名为根用户。尽管根用户通常代表着超级用户账户，但从技术角度讲，用户标识号为 0 就代表着该账户是超级用户，无论账户名是根用户还是其他的名称。

2. ls 命令显示在/tmp 目录下存在一个名为 myfile 的文件，其所有者是 ellie。只有用户 ellie 才有权限读这个文件。其他权限都被关掉了。

3. 因为这是一个超级用户账户，所以无需拥有文件的读权限也可以显示文件内容。

16.3 使用 su 命令变为超级用户

su(切换用户)命令使用户无需注销就可以变为其他用户。su 命令使用其他用户名作为参数，如果您知道这个用户的密码，那么系统将启动属于你所指定的用户的一个新 shell，从而可以临时获得这个用户的权限。如果不指定用户名，su 命令默认切换为根用户，然后询问口令。如果在 su 命令(无论是否加用户名)之后使用长划号(–)，则新的 shell 将会继承指定用户的环境，包括 SHELL、HOME 和 PATH 等变量的设置。

通常情况下，当系统管理员希望运行一些需要根用户权限的命令时，首先使用一个非特权账户登录，然后再使用 su 命令切换到超级用户。一个新的根用户 shell 将被启动，在运行完特权命令(假设是用于解决一个普通用户不能处理的问题)之后，超级用户将会从根用户 shell 中退出并返回到非特权用户。

范例 16-2

```
1  $ id  # or use whoami instead
   uid=501(ellie) gid=100(users) groups=100(users)
2  $ echo $PATH
   /usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:.
```

```
3  $ su
   Password:
4  # id
   uid=0(root) gid=0(root) groups=0(root)
5  # echo $PATH
   /usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:.
6  # exit
   exit
7  $ su -
   Password:
8  # id
   uid=0(root) gid=0(root) groups=0(root)
9  # echo $PATH
   /usr/sbin:/bin:/usr/bin:/sbin:/usr/X11R6/bin
```

说明

1. id 命令显示 ellie 是当前用户。

2. 显示了用户 ellie 的 PATH 变量。

3. 运行 su 命令以将用户切换到根用户。注意必须输入密码。

4. 提示符改为#号。这个符号总是代表着一个超级用户。运行 id 命令，显示当前 shell 具有根用户权限。

5. 显示的 PATH 变量值与之前的相同。这是因为简单 su 命令(不带长划线)用于切换到 root 用户，它并不改变 PATH 变量的值。

6. 运行 exit 以终止由 su 命令创建的 shell。

7. 提示符变成之前的美元符号，这是因为前面一个命令终止了属于根用户的 shell。再次运行 su 命令，这次带有长划号。

8. id 命令证实 root 为当前用户。

9. 因为这次运行 su 命令时，以长划号为选项，所以将使用根用户的 PATH 变量的值。当 PATH 变量的值被回显时，它显示 root 的路径。

如果要使用 su 命令以获取根用户特权，应该使用长划线(-)选项以继承根用户环境，尤其是根用户的 PATH 变量。下面的例子显示了带长划线和不带长划线两种情况下运行 su 命令的结果。注意，在不带长划线时，运行的 useradd 命令是错误版本。

范例 16-3

```
1  $ id
   uid=501(ellie) gid=100(users) groups=100(users)
2  $ echo $PATH
   /usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:.
3  $ su
   password:
4  # echo $PATH
   /usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:.
5  # cd /tmp
6  # ls -l useradd
   -rwxr-xr-x 1 ellie group 65 Feb 12 11:56 useradd
```




```
7  # ls -l /usr/sbin/useradd
   -rwxr-xr-x   1 root    root      57348 Mar 17  2003
   /usr/sbin/useradd
8  # useradd newguy
   this is the useradd script in the /tmp directory
9  # tail -1 /etc/passwd
   ellie:x:501:100::/home/ellie:/bin/bash
   # exit
10 $ su -
   password:
11 # useradd newguy
12 # tail -1 /etc/passwd
   newguy:x:502:100::/home/newguy:/bin/bash
13 # exit
```

说明

1. 运行的 id 命令显示当前 shell 属于非超级用户 ellie。

2. 显示了 PATH 变量，它包含.目录，但并不包含/usr/sbin 目录。

3. 运行 su 命令，启动一个新的根用户 shell。

4. 显示了 PATH 变量，其值并未改变。

5. 运行 cd 命令，/tmp 成为当前目录。

6. ls 命令显示当前目录/tmp 中存在一个名为 useradd 的程序。一般的系统中并没有这个文件，当前系统上的/tmp 目录中存在这个程序纯属偶然。

7. ls 命令显示在/usr/sbin 目录下也有一个名为 useradd 的程序。该目录下的这个标准的 useradd 命令用于系统管理员创建新的账户。

8. 运行 useradd 命令。因为.(点)目录在 PATH 变量中，所以此时 shell 运行的是当前目录下的 useradd 程序，而这并不是用户的真实意图，用户实际希望运行的是/usr/sbin 目录下的 useradd 命令。因为这个错误，导致账户 newguy 并未加入到系统中去。

9. 当前根用户 shell 被 exit 命令终止。

10. 使用长划线，并再次运行 su 命令。su 命令将使用根用户的 PATH 变量取代被创建的新 shell(su 命令创建的 shell)之前的非特权用户的 PATH 变量。

11. 再次执行 useradd 命令。shell 将使用根用户的 PATH 来查找命令，最后将执行/usr/sbin 目录下的 useradd，这个命令将在/etc/passwd 文件的末尾加入一个新行以创建这个账户。

12. tail 命令显用于显示文件/etc/passwd 的最后一行。可以看到 newguy 账户已经被加入。

13. 根用户 shell 退出。这是一个良好的习惯—当超级用户完成了需要根用户权限的命令后，根用户 shell 应该被终止。如果超级用户忘了从根用户 shell 退出，则任何用户都可以使用这个账户。这可不是个好主意！

16.3.1 以根用户身份运行脚本

从根用户(超级用户)账户运行脚本与从普通用户运行脚本有些细微的差别。一般情况

下，根用户的 PATH 变量与非特权用户的 PATH 变量的值是不同的。根用户的 PATH 变量通常包含有系统命令所在的目录。例如，目录/usr/sbin 通常是根用户 PATH 的一部分，但却不是根用户以外账户的 PATH 变量的一部分。为防止根用户不慎运行了当前目录下的程序，根用户的 PATH 变量通常并不包含当前目录，也就是由一个句点代表的点目录。

运行当前目录下的脚本 因为“.” (点目录)不是根用户 PATH 变量的一部分，因此以常规方式在命令行键入脚本名并不能调用脚本执行。如果以根用户身份运行一个脚本，应该在脚本前加上调用 shell 的名字。

范例 16-4

```
1  # cat myscript
   #! /bin/sh
   echo this is my script
2  # echo $PATH
   /usr/sbin:/bin:/usr/bin:/sbin:/usr/X11R6/bin
3  # myscript
   bash: myscript: command not found
4  # /bin/sh myscript
```

说明

1. 显示了脚本 myscript 的内容。脚本的第一行指示它是为/bin/sh 也就是 Bourne shell 程序而编写的。

2. 显示了 PATH 变量的值。注意 PATH 变量并不包含点目录(.)。

3. 因为 PATH 变量不包含.(点)目录，这个命令失败。

4. 将脚本名作为一个参数传递给 shell：/bin/sh，这个命令成功执行。以这种方式调用 shell，将不会使用 PATH 变量来定位脚本，而改为在当前工作目录查找该脚本。

在当前工作目录执行脚本或其他命令的另一种方式是在程序前加上./前缀以表明该程序位于当前工作目录。

范例 16-5

```
1  # cd /usr/local/bin
2  # myprog
   -bash: myprog: command not found
3  # ./myprog
   this is my program
```

说明

1. 当前目录切换至/usr/local/bin。

2. 这个目录包含一个名为 myprog 的程序。试着运行 myprog 程序但未能成功。因为 PATH 中没有.目录，因此 shell 无法定位该程序。

3. 指定 myprog 程序的位置(.目录)后，命令成功运行。

16.3.2 以 root 身份运行的脚本(setuid 程序)

运行 setuid 程序的用户将临时成为程序的拥有者，以获得相同的执行权限。尽管大多



数的 setuid 程序设置为根用户，它们也可以设置为其他任何用户。如果一个 setuid 程序被设置为根用户，则当一个非特权用户运行这样的程序时，将临时变为根用户。这种情况可以形象地比作灰姑娘的故事，当程序退出时，用户返回到它原来的身份并失去了所有的根用户特权。passwd 程序是一个很好的 setuid 程序的例子。当您更改密码时，你会临时地变成根用户，但这仅仅是当执行 passwd 程序时。这就是为什么您能够更改自己的密码而不用向系统管理员求助的原因。如果不能以 root 身份运行的话，您将不能够访问普通用户所不能访问的/etc/passwd(或/etc/shadow)文件。使用 ls -l 命令将能够识别出 setuid 程序。如果程序是 setuid 程序，则 x(执行)权限将被替换为 s。

范例 16-6

```
1   $ ls -l /bin/passwd
    -r-sr-sr-x   1 root    sys       21964 Apr  6  2002 /bin/passwd
2   $ ls -l /etc/shadow
    -r--------   1 root    sys       4775 May  5 13:33 /etc/shadow
```

说明

1. ls -l 的输出显示 setuid 程序将在根用户权限下运行。字母 s 替换了权限域的 x(执行位)。

2. 保存密码的/etc/shadow 文件(Solaris)，显示它是根用户所有的文件，仅根用户可以读取。

shell 程序可以写成 setuid 程序，如果系统管理员没有注意这一点的话，它们将是一个严重的安全隐患。如果一个脚本是 setuid 程序(设置为根用户)，则派生的 shell 也是一个根用户所有的 shell，脚本中命令以 root 身份执行。为什么有些脚本需要是 setuid 程序呢？如果一个数据库或日志文件包含不能被普通用户访问的信息，则该文件的权限必须对除文件拥有者之外的人关闭。如果某个脚本需要访问该文件，而该脚本是由非特权用户执行的，那么脚本将会失败并显示 Permission denied 错误。如果脚本是一个 setuid 脚本，则运行脚本的用户将会获得文件拥有者的身份从而可以访问文件中的数据。并不是所有的 shell 都允许将脚本 setuid 设为根用户权限，即使 shell 允许这样做，操作系统可能也不会允许。例如，bash 不支持 setuid 脚本，尽管 Korn shell 有一种特权模式(-p)可以运行 setuid 脚本，但这仅在操作系统允许的情况下才能使用。下面是将一个 C/TC shell 脚本设置为 setuid 程序的步骤。

1. 脚本第一行是

```
#!/bin/csh -feb
```

在-feb 选项中：
-f 指快速启动。不执行.cshrc 文件
-e 指当被中断时立即中止
-b 指出这是一个 setuid 脚本

2. 下一步，更改脚本的权限以使得它能够作为一个 setuid 程序运行

```
% chmod 4755 script_name
或
% chmod +srx script_name
```

```
% ls -l
-rwsr-xr-x   2 ellie       512 Oct 10 17:18 script_name
```

因为 setuid 程序有一定的安全隐患，大多数系统允许管理员为单独的文件系统禁用 setuid 程序和 setgid 程序。UNIX 命令 find 可用来定位 setuid 为根用户的程序：

```
# find / -user root -perm -4000 | more
```

在脚本内部，可以使用所有 shell 都具有的文件测试操作符来检查 setuid 程序文件。例如，Korn shell 与 bash shell，其测试为：

```
if [[ -u filename ]] ; then
```

C/TC shell:

```
if ( test -u filename ) then
```

Bourne shell:

```
if test -u filename; then
```

或

```
if [ -u filename ]; then
```

16.4 引导脚本

在系统引导阶段，shell 脚本用来执行一些诸如启动监护进程和网络服务、启动数据库引擎、加载磁盘等任务。同时，它们也用于系统关闭阶段。因为尚未有统一的名称，不同的文献和不同的人对这些脚本的称谓各有不同：引导脚本、启动与关闭脚本、初始化脚本或运行控制脚本。使用最频繁的是称为引导脚本，本章其余的部分将用这个词代指这些脚本。在几乎所有的 UNIX/Linux 系统上，引导脚本都是用 Bourne shell 或 bash shell 编写的。UNIX 系统的安装将首先从一系列引导脚本开始，系统管理员不必亲自编写这些脚本，但必须能够调试和修改这些脚本。

对那些熟悉 System V 和 BSD 风格的 UNIX 系统之间差别的人来讲，他们会注意到本节集中在 System V 风格的引导脚本上，这是因为这种风格的脚本是最常用的。

16.4.1 相关术语

在本文中，我们使用了与 shell 相关的一些词，如内核、初始化进程、进程标识号、监护进程和初始化脚本等。因为这些词与引导进程、系统初始化脚本相关，因此本节将对这些词做进一步的分类。

内核与 init 进程　系统引导时，将从磁盘加载 UNIX/Linux 内核。内核用于管理操作系统从引导到关闭的整个过程。首先它初始化设备驱动器，启动交换进程，加载根文件系统(/)。然后内核创建系统中的第一个进程，称之为 init 进程。该进程是所有进程的父进程，



其 PID(Process Identification Number，进程标识号)为 1。init 进程启动后，它从初始化文件/etc/inittab 中读取信息。该文件定义了引导阶段及常规操作阶段需启动的进程，串口上登录的监管进程，以及用于决定启动哪些进程或进程组的运行级。

运行级 运行级也称为系统状态，决定了系统当前可用的进程集合。通常有 8 个运行级：0~6、s 或 S。运行级分为 3 类：暂停、单用户和多用户。在系统状态为暂停时，UNIX 并未运行，它被暂停。系统转向暂停状态将关闭系统。运行级 0 是暂停状态。在某些版本的 UNIX 系统上，特别是 Solaris，运行级 5 也是一个暂停状态，与运行级 0 不同的是，当 UNIX 系统暂停后机器将自动切断电源。单用户运行级为 S 或 1。在单用户模式中，系统控制台被打开并且仅有根用户登录(参见稍后的“单用户模式”)。多用户模式的运行级为 2~5。多用户模式允许用户登录系统，其具体的运行级号根据 UNIX/Linux 系统版本的不同而有很大差别。更复杂的是，在某个特定的系统上，多用户运行级可能会多于一个。例如，在 SuSE Linux 系统上，运行级 2~5 均为多用户运行级。运行级 2 仅允许用户登录一个字符界面、单屏幕会话。运行级 5 允许登录并启动包括诸如 KDE(K Desktop Environment)的窗口管理器等其他进程。超级用户可以使用 init(或 telinit)命令切换到一个不同的运行级。例如，init 0 将系统切换到运行级 0，也就是关闭系统。

因为每个生产商对各个运行级都有不同的定义，因此运行级编号会比较混乱。它们唯一一致的地方在于都使用运行级 0 表示暂停状态，运行级 2 和 3 表示多用户状态。命令 who –r 将列出当前的运行级。参见表 16-1。

表 16-1 运行级

运 行 级	含 义
Solaris	
S、s	单用户模式。加载基本系统操作所需的文件系统
0	暂停
1	系统管理员模式。加载所有的本地文件系统。运行少量必需的系统进程。也是一种单用户模式
2	系统进入多用户模式。创建所有的多用户环境终端进程与监护进程
3	通过让本地资源在网络上可用来扩展多用户模式
4	可以定义为另一种可选的多用户环境配置。它不是系统操作所必需的，通常并不使用
5	关闭机器从而可以安全地切断电源。如果有可能还可以直接将机器电源切断
6	重启动
a、b、c	仅处理在/etc/inittab 中有 a、b 或 c 运行级的项。这里面有一些伪状态，它们定义了一些要运行的特定命令，但并不导致运行级发生改变。
Q、q	重执行/etc/inittab
HP-UX	
0	系统被彻底关闭
1、s、S	单用户模式。所有的系统服务和监护进程被终止，且所有的文件系统被卸载

(续表)

运 行 级	含 义
2	多用户模式，但不开启 NFS
3,4	多用户模式，开启 NFS
4	多用户模式，开启 NFS 并使用 HP 的桌面
6	重启动
OpenBSD	
-1	永久不可靠模式——系统一直运行在级别 0 模式
0	不可靠模式——所有的设备都有读或写权限
1	可靠模式——加载文件系统的磁盘/dev/mem 和/dev/kmem 是只读的
2	高可靠模式——与可靠模式相同，同时无论是否被加载，磁盘总是只读的并且系统调用 settimeofday 仅可以向前更改时间
Linux	
0	将系统挂起
1	单用户模式
2、3	多用户模式，通常是相同的。级别 2 或级别 3 是默认设置
4	未使用
5	图形环境(X Windows 系统)的多用户模式
6	重启动系统

范例 16-7

```
$ who -r
.     run-level  3  Mar 18 14:24    3       0  S
```

说明

命令 who -r 显示 init 进程的当前运行级，其结果是运行级 3，即多用户模式(Solaris)。

单用户模式 在单用户模式，UNIX 系统已运行，但用户不能够登录。init 1 或 init s 都会将系统转换到单用户状态。与系统交互的唯一方式是通过在特定窗口或工作站自动启动属于根用户的 shell。单用户状态一般用来进行系统维护，系统进程并不会全部被启动。例如，网络与数据库进程就不会被运行。通常，系统管理员将系统切换到单用户模式以修复某个严重的软件问题、安装软件以及执行一些其他任务来防止用户见到功能不完全的系统或干涉到系统的运行(例如，在升级系统时不希望用户访问文件)。

引导脚本 系统上定义的每种运行级都会包含一个脚本目录，这些脚本运行在特定的运行级上，它们管理诸如 mail、cron、网络服务、lpd 之类的服务。从目录名可以看到运行级，如目录 rc5.d 包含运行级为 5 的脚本，目录 rc3.d 包含运行级为 3 的脚本，以此类推。根据 UNIX/Linux 版本的不同，这些目录位于不同的位置。例如，运行级为 3 的脚本通常存储在下面某个目录中：/etc/rc3.d、/sbin/rc3.d、/etc/init.d/rc3.d 或/etc/rc.d/rc3.d。一旦找到了正确的目录，各种版本的 UNIX/Linux 就使用通用的机制对这些目录中的脚本命名。脚



本名以字母 S 或 K 开头，后跟一个数字，然后是一个描述脚本任务的名称。脚本的参数是 start 或 stop。参数 start 要求脚本启动服务，参数 stop 则要求脚本关闭服务。如果脚本名以 S 开头，则该脚本的参数为 start，如果脚本名以 K 开头，则脚本的参数为 stop。K 类型的脚本在系统将切换到低运行级时特别有用，因为系统运行级降低，所以需要使用它来关闭进程。脚本按 ls 命令列出的顺序运行。

范例 16-8

```
$ cd rc3.d
$ ls
README          S34dhcp      S77dmi        S89sshd      s15nfs.server
S13kdc.master   S50apache    S80mipagent   S90samba     s99idled
S14kdc          S52imq       S81volmgt     S99fixkde
S16boot.server  S76snmpdx    S84appserv    S99snpslmd
```

说明

这个范例来自运行 Solaris5.9 的系统。其中显示了目录 rc3.d(运行级 3)的内容。可以用来核对您所在系统上的这些文档。

范例 16-9

```
$ cd rc3.d
$ ls
K001inuxconf  K25squid     K55routed   K92iptables      S55sshd
K03rhusd      K28ams       K61dap      K95kudzu         S56rawdevices
K05anacron    K30mcserv    K65identd   K95reconfig ig   S56xinetd
K05innd       K30sendmail  K65kadmin   K96irda          S601pd
     <not all output is shown>
```

说明

这个例子来自运行 Red Hat Linux 的系统。显示了目录 rc3.d(运行级 3)的部分内容。可用来核对您所在系统上的这些文档。

监护进程　监护进程(daemon)[①]是在后台运行的进程，代表用户执行一个或多个任务。例如，打印机监护进程，即 lpsched，每次向打印机发送一个文件。如果使用 lpsched 监护进程进行打印的系统没有运行该进程，则文件将不会被发送到打印机。Web 服务器，如 Apache 就是一个监护进程。它会保持在睡眠状态直到出现 Web 页请求。cron 程序(有些系统上称为 crond)也是一个监护进程。它每分钟检查一次 cron 指令文件(称为 crontab 文件)，以查看是否要执行文件中指定的某些命令。一旦到了指定的时间，cron 监护进程将自动运行 crontab 文件中指定的程序。下面的“一个引导脚本的例子——cron 工具”对 cron 工具进行了详细的讨论。很多监护程序，如 cron，在系统引导的过程中被启动，在系统运行的整个期间一直运行。其他的监护程序，如 telnet 在主监护进程的控制之下，仅在需要的时候被启动(主监护进程需要一直运行)。

① 这个词源自希腊神话。daemon 指的是守护者的灵魂，不要将它与 demon()混淆，后者与魔鬼等邪恶的事物相关。

16.4.2 一个引导脚本的例子——cron 工具

在本节，我们将看到一个 Red Hat Linux 中的引导脚本范例。该脚本在系统引导时启动 cron 监护进程。同时，该脚本也可以手工运行以控制监护进程。cron 引导脚本是一个很好的范例，这是因为它非常容易理解，并且几乎所有版本的 UNIX 系统上都有 cron 程序。在查看 cron 引导脚本之前，弄清楚 cron 程序究竟做什么事情是非常有帮助的。根用户及其他用户可以使用 cron 工具来按预先设置的时间调度运行系统命令、shell 脚本、程序等。用户无需登录即可与系统交互，cron 会自动运行用户 crontab 文件中的命令。该文件包含一个表，指定了需要运行的命令及其运行日期、时间。下面的范例显示 Solaris cron 监护进程以 root 身份运行。

范例 16-10

```
$ ps -ef | grep cron | grep -v grep
root   202     1  0   Mar 18 ?       0:03 /usr/sbin/cron
```

Red Hat Linux 在引导系统时启动 crond 监护进程

```
$ ps aux  | grep cron | grep -v grep
root    436 0.0  0.5 1552  700   ?    S  06:13  0:0   crond
```

创建 cron 指令文件(crontab 文件) cron 监护进程读取由系统用户提交的 crontab 文件。文件的每一行都是由空格隔开的 6 个域。前 5 个域设置的是 cron 监护进程运行第 6 个域中所指定命令的时间。前 5 个域的值为：分钟(00~59)，小时(00~23)，日期(1~31)，月份(1~12)和星期(0~6)。第 6 个域指定的是在前 5 个域指定时间要运行的命令。参见表 16-2。

表 16-2 crontab 文件各域的值

域	值
分钟	00~59
小时	00~23
日期	1~31
月份	1~12
星期	0~6，0 代表星期天(Linux 使用 sun、mon 等)

前 5 个域也可以使用下列任一种格式。

- 星号(*)匹配该域的所有值
- 单个整数即匹配准确值
- 由逗号隔开的一列整数(例如，1，3，5)匹配其中的任何一个。由长划线隔开的整数范围(如，4-6)匹配该范围内的值。

下面的范例示意了如何使用 crontab 命令向 cron 提交一个 crontab 文件。

范例 16-11

```
1   # crontab -l > instructions
2   # vi instructions
```



```
   .
   (There might or might not be lines at the top of this file.
    You can ignore any lines except the one added below.)
3  25 1 * * * /root/checkpercent
4  # crontab instructions
5  # crontab -l
   .
   (There may not be lines above the entry that was added in item 3.)
   .
   25 1 * * * /root/checkpercent
```

说明

1. 命令 crontab -l 的输出重定向到文件 instructions 中。在这个例子中，因为 crontab 文件中并没有任何命令，所以 instructions 文件仅包含一些由 crontab 命令产生的注释。

2. 使用 vi 编辑 instructions 文件。该文件可能包含或不包含用于启动的行，它取决于您所使用的 UNIX/Linux 版本和根用户之前是否为 cron 输入过指令。

3. 在文件尾加入了一条新指令。该指令通知 cron 在每月每天的上午 1：25 运行 /root/checkpercent 脚本。该文件中指定时间的详细信息(时间由行首的数字和星号指定)可参考 crontab 帮助。

4. crontab 命令用于向 cron 提交 instructions 文件。

5. crontab -l 命令使 cron 显示当前用户(本例中是根用户)的当前指令集合。这是用于检验 cron 是否接受了先前命令中提交的文件。

cron 引导脚本　在查看 cron 引导脚本的具体内容前，先看看引导脚本的一般布局。下面这个例子显示了 Red Hat Linux 系统上运行级为 5 的引导脚本的部分列表。尽管大多数其他的 UNIX 版本对引导脚本有同样的布局，一些系统如 Mac OS X 中的 Darwin，其布局却有相当大的差别。

下面是 Red Hat Linux 系统上删节后的目录列表，它包含运行级为 5 的引导脚本。

范例 16-12

```
1  # pwd
   /etc/rc5.d
2  # ls
   .     K20nfs    K45named    S55sshd     S90crond
   ..    K35smb    S12syslog   S80sendmail
3  # ls -l /etc/rc5.d/*cron*
   lrwxrwxrwx 1 root 15  Oct 13 21:19 /etc/rc5.d/S90crond ->
   ../init.d/crond
4  # ls -l /etc/init.d/cron*
   -rwxr-xr-x 1 root 1316 Feb 19 2004 /etc/init.d/crond
```

说明

1. 当前目录为/etc/rc5.d，运行级为 5(Red Hat Linux)的引导脚本目录。

2. 列出的结果显示有些脚本名以 S 开头，有些以 K 开头。

3. 该脚本启动 cron 监护进程，因为它的文件名以 S 开头。如果该脚本名以 K 开头，则该脚本将关闭 cron 监护进程。输出显示该文件实际上是一个链接——文件权限前的字母

l 表示一个链接。字符->指向/etc/init.d 目录中的被链接的 crond 文件。

4. 运行级中的每个脚本通常是/etc/init.d 目录中原件的一个链接。

5. 列出了 cron 引导脚本的原件。注意，有些原件名不以 S 或 K 开头，有些并不包含数字。它们通常是引导目录中源脚本的多重链接，仅第一个链接名才包含数字。源脚本用于启动或关闭 cron 监护进程。下一节我们将详细讨论这个脚本。

cron 引导脚本，称为 crond，可以手工运行以启动或关闭 cron 监护进程。在有些版本中，它还可以用于检查 cron 的状态或重启 cron，还有些版本使用更多的参数。按惯例，一般是在脚本名前加上编写该脚本所使用的 shell 名从而运行引导脚本。下面这个范例运行了/etc/init.d 目录下的 crond 脚本的原件。因为有文件链接到这个文件，所以运行/etc/rc5.d/crond 脚本可以达到相同的效果。

范例 16-13

```
1   # cd /etc/init.d
2   # ls crond
    crond
3   # sh crond stop
    Stopping crond:   [OK]
4   # sh crond start
    Starting crond:   [OK]
5   # sh crond restart
    Stopping crond:   [OK]
    Starting crond:   [OK]
```

说明

1. cd 命令切换目录至包含引导脚本原件的/etc/init.d 目录。

2. ls 命令检验 cron 引导脚本是否存在。

3. 以 stop 为参数运行 crond 脚本以关闭 cron 监护进程。因为根用户是当前用户，所以通过在脚本名前加上编写该脚本所使用的 shell 名从而运行该脚本。

4. 脚本再次运行，这次启动 cron 监护进程。

5. 脚本第 3 次被运行。这次先关闭监护进程，然后再启动。

下面是 Red Hat Linux 上的 cron 监护进程引导脚本的简化版本。该脚本使用另一个称为 functions 的脚本，functions 脚本定义了引导脚本的 daemon 和 killproc。daemon 函数以一个监护进程名为参数，试着启动该监护进程，并根据执行情况返回一个成功或失败值。

范例 16-14

```
#! /bin/bash
# crond Start/Stop the cron clock daemon.
#
# chkconfig: 2345 90 60
# description: cron is a standard UNIX program that runs
# user-specified programs at periodic scheduled times.
# config: /etc/crontab
# pidfile: /var/run/crond.pid
```

```
# Source function library.
1   . /etc/init.d/functions
2   RETVAL=0 ; prog="crond"
3   start() {
4       echo -n $"Starting $prog: "
5       daemon crond
6       RETVAL=$?
7       echo
8       [ $RETVAL -eq 0 ] && touch /var/lock/subsys/crond
9       return RETVAL
    }
10  stop() {
11      echo -n $"Stopping $prog: "
12      killproc crond
13      RETVAL=$?
14      echo
15      [ $RETVAL -eq 0 ] && rm -f /var/lock/subsys/crond
        return RETVAL
    }
16  restart() {  stop ; start }
17  case "$1" in
18      start)   start          ;;
19      stop)    stop           ;;
20      restart) restart        ;;
21      *)      echo $"Usage: $0 {start|stop|restart}"
        exit 1          ;;
esac
```

说明

1. 使用一个名为 functions 的脚本，其中定义了现在脚本中使用的各个函数。

2. 变量 RETVAL(保存函数返回值)被初始化为 0，变量 prog 被初始化为 crond。

3. 从这行开始定义 start()函数。

4. 显示一条消息，说明 cron 监护进程已经启动。

5. daemon 函数以 crond 为参数被调用(该函数在脚本开始处加入的 functions 脚本中定义)。该函数尝试启动 crond 监护进程并返回一个成功(0)或失败值(1)。

6. 变量 RETVAL 被赋值为状态变量？的值。它包含 daemon()函数的返回值。操作系统在引导时使用这个值向屏幕打印 OK 或 FAILED 消息。

7. echo 命令向屏幕显示一空白行。

8. 如果返回值显示成功(0 表示成功)，则用 touch 命令为 cron 监护进程创建一个锁文件/var/lock/subsys/crond。

9. 函数返回 RETVAL 变量的值。

10. 这一行开始了 stop()函数的定义。

11. 一条消息显示 cron 监护进程已经被停止。

12. 调用 killproc 函数，它在脚本第一行引入的函数库中有定义。该函数以 crond 为参数，尝试关闭监护进程并基于其执行成功或失败的情况返回一个值。

13. RETVAL 变量被赋值为 killproc()函数的返回值，该返回值存储在$?中。
14. 输出一空白行。
15. 如果返回值显示成功，删除 cron 监护进程的锁文件/var/lock/subsys/crond。
16. 定义用于先关闭再启动 cron 监护进程的 restart()函数。
17. case 命令对$1 求值并保存该脚本第一个参数的值：start 或 stop。
18. 如果$1 的值为 start，则运行 start()函数。
19. 如果$1 的值为 stop，则运行 stop()函数。
20. 如果$1 的值为 restart，则运行 restart()函数。
21. 其他情况下，显示一条有关用法的消息然后脚本以表示失败的状态值 1 退出。

16.4.3 编写一个可移植的脚本

系统管理员通常需要编写在各种 UNIX 版本上运行的 shell 脚本。因为各种 UNIX 的命令并不相同，所以脚本必须能够确定它将运行在哪个版本的 UNIX 上。使用 uname 命令可以做到这点。下面是一些流行的 UNIX 版本及其 uname 命令的输出。创建一个在这些不同 UNIX 版本之间可移植的脚本，必须检查 uname 的输出并对命令做相应的修改。

UNIX 版本	uname 输出
AIX	AIX
FreeBSD	FreeBSD
HP-UX	HP-UX
IRIX	IRIX
Linux	Linux
Mac OS X	Darwin
NetBSD	NetBSD
OpenBSD	OpenBSD
SCO OpenServer	5 SCO_SV
Solaris	SunOS

通常用 case 命令来检测当前使用的操作系统的类型。范例 16-15 对如何检测 Linux(任何商标)、HP-UX、Solaris、FreeBSD 和 Mac OS X UNIX 给出了示意。

范例 16-15

```
1  uname_out=`uname`
2  case "$uname_out" in
3  HP-UX)  echo "You are running HP-UX"
       ;;
4  SunOS) echo "You are running Solaris"
       ;;
   FreeBSD) echo "You are running FreeBSD"
       ;;
   Linux) echo "You are running Linux"
```



```
        ;;
    Darwin) echo "You are running Mac OS X"
        ;;
5       *)  echo "Sorry, $uname_out UNIX "
            echo "is not supported by this script"
        ;;
6   esac
```

说明

1. 变量 uname_out 被赋值为 uname 命令的输出。它将当前使用的 UNIX 的版本赋给该变量。

2. case 语句对 uname_out 变量求值。

3. 如果对 uname_out 变量求值的结果是 HP-UX，则表示是 Hewlett-Packard 版本的 UNIX 并显示 You are running HP-UX。

4. 如果对 uname_out 变量求值的结果是 SunOS，则显示相应的消息。case 语句在后续语句中检测其他版本的 UNIX。

5. 如果 uname_out 包含有未在上述选项中列出的值，则匹配默认模式(以一个星号表示)。在这种情况下，显示一条通用的消息通知用户操作系统不支持这个脚本。

6. case 语句以 esac 语句结束。

范例 16-16 是一个脚本实例，用于在机器的文件系统饱和时通知系统管理员。虽然可以修改脚本使之符合任意版本的 UNIX，但该脚本现在并不能够支持所有版本的 UNIX。该脚本实例改变 df 命令的形式以适应 4 种类型的 UNIX。

范例 16-16

```
# cat /root/checkpercent
1   #! /bin/sh
2   rm $HOME/df_output $HOME/message 2> /dev/null
3   uname_out=`uname`
4   case "$uname_out" in
5       HP-UX)
6           bdf | awk  '{print $5,$6}' | awk -F% '$1>90 {print $0}' \
                                     > $HOME/df_output
        ;;
7       SunOS)
8           df -k | awk '{print $5,$6}' | awk -F% '$1>90 {print $0}' \
                                     > $HOME/df_output
        ;;
9       Linux)
10          df | awk '{print $5,$6}' | awk -F% '$1>90 {print $0}' \
                                     > $HOME/df_output
        ;;
11      Darwin)
12          df | awk '{print $5,$6}' | awk -F% '$1>90 {print $0}' \
                                     > $HOME/df_output
        ;;
```

```
13     *)
        echo "Sorry, $uname_out UNIX not supported by this script"
        ;;
   esac

14  if [ -s $HOME/df_output ]
   then
15  echo "** WARNING **" > $HOME/message
   echo "The following file systems are filling up." >> $HOME/message
   echo "You may want to look into the situation." >> $HOME/message
   cat $HOME/df_output >> $HOME/message
16  cat $HOME/message
17  echo "This warning message is stored in the file $HOME/message"
   echo "You should create a copy of the file now if you would"
   echo "like to save this message."
   fi
18 rm $HOME/df_output
```

说明

1．shell 名为/bin/sh。在有些系统上，/bin/sh 是 Bourne，其他系统上是 bash 或 POSIX shell——所有这些 shell 都与 Bourne shell 兼容。为证实这是一个可移植的脚本，需使用 Bourne shell 的语法。

2．这个命令在文件$HOME/df_output 和$HOME/message 存在时，删除它们。通过将该命令产生的错误信息重定向到/dev/null 文件从而丢弃它们。删除这些文件是一种预防措施——如果他们任何一个存在，都可能对脚本的运行产生问题。$HOME 是存放用户主目录的变量，因此$HOME/message 指的是运行脚本的用户主目录中一个称为 message 的文件。$HOME/df_output 指的是用户主目录中名为 df_output 的文件。

3．uname 命令的输出被赋给变量 uname_out。注意，与前面的范例一样，该命令使用反引号，而非普通的单引号。

4．case 语句对变量 uname_out 求值，该变量包含操作系统的名称。

5．如果对变量求值的结果为 HP-UX，则 case 语句第 6 行被执行。

6．运行 HP-UX 版本的 df 命令 bdf。bdf 命令的输出用管道传送给 awk，后者选择第 5 个和第 6 个域，也就是满载百分比和文件系统名。然后这两个值又被用管道传送给第二个 awk，它用于显示满载百分比大于 90 的行(只要第一个域大于 90，$0 将打印整行)。输出最后存放在文件$HOME/df_output 中。

7．如果是 Sun 系统则执行第 8 行的命令。

8．运行 Solaris 版本的 df 命令 df -k。其余的命令与 HP-UX 版本的相同。

9．如果是 Linux 系统则执行第 10 行的命令。

10．为 Linux 运行普通的 df 命令。其余的命令与 HP-UX 版本的相同。

11．如果是 OS X 系统则执行第 12 行的命令。

12．运行 df 命令。注意这与 Linux 运行的是相同的命令。因此这两种情况可以合并到一个 case 语句中。

13．如果变量 uname_out 包含其他的值，则显示错误消息。



14. 这个 if 语句查看文件$HOME/df_output 是否包含任何文本。这是一个真(true)/假(false)测试，如果文件$HOME/df_output 包含某些文本则测试结果为真，如果文件为空或根本不存在则为假。如果$HOME/df_output 文件确实存在则执行第 15 行的命令。

15. 产生一条消息并将其存放在$HOME/message 文件中。3 个 echo 命令显示警告信息，然后将$HOME/df_output 文件追加到文件$HOME/message 中。

16. 显示文件$HOME/message 的内容。这将显示一列满载的文件系统的警告信息。

17. 脚本打印$HOME/message 文件中的提示信息。

18. 删除不再需要的$HOME/df_output 文件。

16.4.4 用户指定初始化文件

用户账号创建伊始，系统管理员通常在用户主目录放置一些登录文件如.profile 和.login。系统管理员必须懂得 shell 语法以编写出适合用户的这类文件。

每种 shell 都有一个或多个可以放在用户主目录的初始化文件。作为系统管理员，您可能希望在自己的主目录被创建时先放置这类文件的一个初始版本。大多数版本的 UNIX 能够使用/etc/skel 目录帮助您自动进行这个过程，当然不同版本之间这个目录名稍有差别。当使用 useradd 命令创建新账户时，/etc/skel 中的所有文件被复制到用户主目录中。

范例 16-17

```
1   # ls -a /etc/skel
    . .. .profile .cshrc
2   # useradd -m newguy1
3   # ls -a /home/newguy1
    . .. .profile .cshrc
4   # useradd newguy2
5   # ls -a /home/newguy2
    ls: /home/newguy2: No such file or directory
```

说明

1. /etc/skel 目录包含两个文件：.profile 和.cshrc。

2. -m 选项通知 useradd 命令为新账户创建主目录。默认的该目录为/home/newguy1。useradd 程序自动地从/etc/skel 目录中将两个文件.profile 和.cshrc 复制到/home/newguy1 目录下。

3. ls 命令检验/home/newguy1 目录是否已经包含了.profile 和.cshrc 两个文件。

4. 如果没有-m 选项，useradd 命令将不会为该账户创建一个主目录。从而也不会从/etc/skel 中复制文件。

5. ls 命令显示了 newguy2 没有主目录。

/etc/skel 可能的文件 在为新用户创建主目录时应该在这个主目录中放置一组作为起点的初始化文件。表 16-3 是当前 5 种主要 UNIX 版本使用的初始化文件图表。您可能希望为所有的这些文件创建一个通用版本放置在/etc/skel 目录中。新的账户被创建时这些初始化文件将被复制到用户主目录中，从而给用户一个适合其登录 shell 的初始化文件的起始版本。可参见其他各章对各种 shell 初始化文件的具体描述。

表 16-3 5 类 Shell 使用的初始化文件

	Bourne	Bash	Korn	C	TC
.profile	√	√	√		
.kshrc			√		
.bash_profile		√			
.bash_login		√			
.bashrc		√			
.bash_logout		√			
.login				√	√
.cshrc				√	√
.tcshrc					√
.logout				√	√

16.4.5 系统范围内的初始化文件

系统管理员有责任为各种登录 shell 维护一个系统范围内的初始化文件集合。系统管理员常常执行一些任务如在相应的文件中为 PATH 变量和 MANPATH 变量设置一个初始值。对登录 shell 来讲，/etc/profile 是 Bourne，Korn 和 bash 3 种 shell 的系统范围内的初始化文件。/etc/.cshrc 或/etc/csh.login 是 C shell 和 TC shell(不同的系统上文件名有所不同)系统范围内的初始化文件。参见表 16-4，大多数 UNIX 安装时就带有一个这些文件的初始版本集合，它们能够为特定的系统定制。

表 16-4 系统范围内的初始化文件

shell 类型	文 件 名	适 用 版 本	备 注
/bin/sh	/etc/profile	大多数版本	仅在登录时运行
	/etc/profile.local	SuSE Linux	除/etc/profile 外它也仅在登录时运行，用于本地设置。SuSE 的/etc/profile 文件包含一条注释要求不要修改/etc/profile 文件，而是修改/etc/profile.local 文件
	/etc/login.conf	FreeBSD	仅在登录时运行
/bin/ksh	/etc/profile	大多数版本	仅在登录时运行
	/etc/profile.local	SuSE Linux	除/etc/profile 外它也仅在登录时运行，用于本地设置。SuSE 的/etc/profile 文件包含一条注释要求不要修改/etc/profile 文件，而是修改/etc/profile.local 文件



(续表)

Shell 类型	文 件 名	适 用 版 本	备 注
/bin/bash	/etc/profile	大多数版本	仅在登录时运行
	/etc/profile.local	SuSE Linux	除/etc/profile 外它也仅在登录时运行，用于本地设置。SuSE 的/etc/profile 文件包含一条注释要求不要修改/etc/profile 文件，而是修改/etc/profile.local 文件
	/etc/bash.bashrc	SuSE Linux	每次 shell 调用时均运行
	/etc/bashrc	某些版本的 Linux	当手工启动 bash 时运行。如果您的 UNIX 系统上该脚本没有自动运行，可以直接调用用户本地的 .bashrc 文件。参见稍后的"/etc/bashrc"以获取更多细节
/bin/csh	/etc/csh.login	Linux，FreeBSD，HP-UX，OS X	与 tcsh 相同，仅在登录时运行
	/etc/.login	Solaris	仅在登录时运行
	/etc/csh.cshrc	Linux，FreeBSD，OS X	每次 shell 调用时均运行
	/etc/.cshrc	Solaris	由 tcsh 而不能由 csh 运行，每次运行 tcsh 时均运行该脚本

除系统初始化文件外，用户登录时还会运行本地初始化文件。例如，当一个用户登录 Red Hat 系统上的 TC shell 后，下面列出的文件将依次被引用。前两个文件是系统初始化文件，后两个文件是本地文件。环境变量 HOME 的值是用户主目录的路径。

```
/etc/csh.cshrc
/etc/csh.login
$HOME/.tcshrc
$HOME/.login
```

/etc/profile /etc/profile 文件包含有当用户登录使用 Bourne，Bash 或 Korn 的系统后将自动运行的命令。/etc/profile 的初始版本在安装时提供。系统管理员通常会修改这个文件以使其适合于特定的系统。

范例 16-18

```
# cat /etc/profile
# System-wide environment and startup programs, for login setup
# Functions and aliases go in /etc/bashrc

1  pathmunge () {
2      if ! echo $PATH | /bin/egrep -q "(^|:)$1($|:)" ; then
3         if [ "$2" = "after" ] ; then
```

```
4           PATH=$PATH:$1
         else
5           PATH=$1:$PATH
         fi
      fi
6   }
    # Path manipulation
7   if [ `id -u` = 0 ]; then
8       pathmunge /sbin
9       pathmunge /usr/sbin
10      pathmunge /usr/local/sbin
      fi
11  pathmunge /usr/X11R6/bin after
12  unset pathmunge
    # No core files by default
13  ulimit -S -c 0 > /dev/null 2>&1

14  USER="`id -un`"
15  LOGNAME=$USER
16  MAIL="/var/spool/mail/$USER"
17  HOSTNAME=`/bin/hostname`
18  HISTSIZE=1000

19  if [ -z "$INPUTRC" -a ! -f "$HOME/.inputrc" ]; then
        INPUTRC=/etc/inputrc
    fi
20  export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE INPUTRC
21  for i in /etc/profile.d/*.sh ; do
22      if [ -r "$i" ]; then
23          . $i
        fi
    done
    unset i
```

说明

1. 函数 pathmunge()开始被定义。该函数以一个目录名为参数，其目的是将作为参数的目录赋给 PATH 变量。该函数还有可选的第 2 个参数 after，它将目录追加到 PATH 变量。注意，该函数定义仅仅是将 pathmunge()的代码放入内存，除非被调用，否则函数将为会被激活。

2. 检查将要加入的目录是否已经在用户的 PATH 变量中了。

3. 检查可选参数 after 是否包含在函数调用中。

4. 将目录(pathmunge 函数的第一个参数)加入到 PATH 变量的起始位置。只有当函数调用没有包含可选参数 after 时这一行才被执行。

5. 将目录(pathmunge 函数的第一个参数)加入到 PATH 变量的结尾。只有当函数调用包含可选参数 after 时这一行才被执行。

6. 结束 pathmunge 定义。

7. 如果用户 UID 等于 0(换句话说，如果用户是超级用户)，则继续向下执行。



8. 调用 pathmunge 函数，将/sbin 目录加入到 PATH 变量中。

9. 将/usr/sbin 目录加入到 PATH 变量中。

10. 将/usr/local/sbin 目录加入到 PATH 变量中。

11. 将/usr/X11R6/bin 目录加入到 PATH 的结尾。因为该行不在 if 语句之内，所以它将总是被执行。

12. 从内存中删除 pathmunge()函数的定义。

13. 将主存储器信息转储(coredump)的大小设为 0——这样可以有效地阻止用户产生主存储器信息转储文件。

14. 将 USER 变量设置为用户登录名。

15. 将 LOGNAME 变量设置为与 USER 相同的值。

16. 将 MAIL 变量设置为用户收信的邮箱文件。

17. 将 HOSTNAME 设置为当前系统的主机名。

18. 将历史列表的大小设置上限为 1000 条命令。

19. 如果变量 INPUTRC 尚未赋值并且$HOME/.inputrc 文件不存在，则将 INPUTRC 变量设置为/etc/inputrc。

20. 输出脚本中设置的所有的变量。

21. 这个 for 循环将遍历/etc/profile.d 目录中的所有文件。

22. 如果用户有当前 for 循环所处理的文件的读权限，则继续向下执行。

23. 引用该文件。

/etc/bashrc　有些版本的 UNIX 系统如 Red Hat Linux 在 bash shell 运行时使用/etc/bashrc 文件运行 bash 命令。该文件包含了一些不能够自动传递给子进程的设置，如命令别名。

范例 16-19

```
1   $ cat /etc/bashrc
2   alias ls="ls -F"
3   alias grep="grep -i"
```

说明

1. 显示了文件/etc/bashrc 的内容。
2. 为 ls 命令定义一个别名 ls –F。当键入 ls 时，将执行 ls –F，从而改变 ls 的输出。
3. 为 grep 命令定义一个别名 grep –i。当使用 grep 命令时，它将是大小写敏感的。

如果希望使用/etc/bashrc 但您的 UNIX 并不自动引用该文件，则可以在用户主目录中的用户私有文件.bashrc 中添加它。

范例 16-20

```
1   $ cat .bashrc
2   if [ -f /etc/bashrc ]
        then
3       . /etc/bashrc
    fi
alias dir=ls
```

说明

1. 引用用户主目录中的.bashrc 文件。

2. 如果/etc/bashrc 文件存在并且是一个普通文件，则跳转到第 3 行。有些机器能够自动运行/etc/bashrc，而在这些机器上该 if 语句是不需要的。

3. 点命令引用/etc/bashrc 文件(在当前 shell 进程的上下文中运行该文件中的命令。也就是说，不创建新的子进程)。

4. 为 dir 命令定义了一个别名。注意，这是由用户.bashrc 文件创建的私有别名。

/etc/csh.login 在登录时，C shell 和 TC shell 会引用/etc/csh.login 文件。范例 16-21 是一个来自 Red Hat Linux 系统的/etc/csh.login 文件。与.login 文件类似，该文件由登录 shell 在登录时运行，而不会被其后派生的任何子进程运行。它用于设置 shell 环境变量和由登录 shell 执行的其他命令。

范例 16-21

```
    # cat /etc/csh.login
    # System-wide environment and startup programs, for login setup
1   if ($?PATH) then
2      setenv PATH "${PATH}:/usr/X11R6/bin"
    else
3      setenv PATH "/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin"
    endif

4   limit coredumpsize unlimited

5   setenv HOSTNAME `/bin/hostname`
6   set history=1000
7   if ( -f $HOME/.inputrc ) then
8      setenv INPUTRC /etc/inputrc
    endif
9   if ( $?tcsh ) then
10     bindkey "^[[3~" delete-char
endif
```

说明

1. 如果 PATH 变量已经被设置，则?的值为真。

2. 将/usr/X11R6/bin 加入到环境变量 PATH 中。

3. 否则，PATH 变量被赋值为 bin:/usr/bin:/usr/local/bin:/ usr/X11R6/bin。

4. limit 命令为主存储器信息转储的大小设置上限。

5. 变量 HOSTNAME 被赋值为 hostname 命令的输出，也就是系统的主机名。

6. 历史列表的大小被设置为最多 1000 条命令。

7. .inputrc 文件设置编辑模式、按键绑定以及 Readline 库。该行检查$HOME/.inputrc 文件是否存在。

8. 如果$HOME/.inputrc 文件不存在，则将 INPUTRC 变量设为/etc/inputrc。

9. 如果当前 shell 为 tcsh，则?将返回真并执行第 10 行。

10. bindkey 命令为用于命令行编辑的删除(Delete)键设置一个转义序列。



/etc/csh.cshrc　当 C shell 和 TC shell 运行时，它们都将引用/etc/csh.cshrc 文件。登录时运行 C 或 TC shell 脚本可以启动 C shell 和 TC shell(如果已经将它指定为用户登录 shell)，或者打开一个运行 C 或 TC shell 的窗口也可以启动 C shell 和 TC shell。下面给出的/etc/csh.cshrc 文件示例是 Red Hat Linux 系统上的。/etc/csh.cshrc 文件应该包含一些不输出给子进程的项，如 umask 的设置或提示符的设置。注意，在 C shell 和 TC shell 中提示符变量是局部的，也就是说它将不会传递给子进程。

范例 16-22

```
# cat /etc/csh.cshrc
1   umask 022
2   if ($?prompt) then
3      if ($?tcsh) then
4         set prompt='[%n@%m %c]$ '
       else
5         set prompt=\[`id -nu`@`hostname -s`\]\$\
       endif
    endif
6   if ( -d /etc/profile.d ) then
7      set nonomatch
8      foreach i ( /etc/profile.d/*.csh )
9         if ( -r $i ) then
10           source $i
          endif
       end
11     unset i nonomatch
    endif
```

说明

1. 将 umask 设置为 022，新创建文件的(组及其他用户的)写权限将被屏蔽。

2. ?用来检查提示符变量是否已经有值。如果有，则该 shell 为一个交互式 shell。

3. ?用来查看 tcsh 变量是否已经有值。该变量仅当处于 tcsh 时才被设置，C shell 中不设置。

4. 如果是 TC shell，则提示符被设置为用户名(%n)后跟一个@符号，再加上主机名(%h)，最后是当前目录(%c)。

5. 如果是 C shell，则提示符被设置为用户名(命令 id －nu 的输出)后加一个@符号，再加上主机名(命令 hostname -s 的输出)，最后是一个美元符号。

6. 如果/etc/profile.d 目录存在，则跳转到第 7 行。

7. nonomatch 设置后，当 shell 通配符不能匹配到一个文件名时，将不产生任何错误消息。该设置将保持有效直至运行 unset 命令。

8. 该 foreach 循环将遍历/etc/profile.d 目录中的所有文件，对名字以.csh 结尾的文件，会依次将它们的名字赋给变量 i。注意，如果*.csh 不能够匹配/etc/profile.d 目录中的任何文件，则通常将会显示错误消息 No match。因为这里设置了 nonomatch，所以不会出现这种情况。

9. 如果运行该脚本的用户对变量 i 中指定的文件有读权限，则跳转到第 10 行。

10. 文件被引用。

11. nonomatch 变量被复位。这意味着当 shell 通配符不能匹配任何文件名时，会再次打印错误消息。变量 i 也被复位，也就是从内存中删除它的值。以常规方式运行的 shell 脚本，也就是通过在命令行中键入脚本名来运行的脚本，将在子 shell 中运行。这样的话，它无需将变量复位。因为当脚本退出后，子 shell 也会退出，于是变量将自动地从内存中删除。因为该脚本被引用，因此并未创建子 shell，变量依然保留在当前 shell 的内存中。而这些变量已经不再需要，因此它们应该被复位。

16.5 小结

作为系统管理员，您是超级用户，拥有至高无上的权力。就好像上帝一样，只要敲击键盘就能创建或销毁系统。然而，责任越大，对知识的一知半解越显得危险。那么，了解 shell 是如何工作的，怎样读、写、修改 shell 脚本对您至关紧要。因为它不仅能够自动运行每天任务，还要为在系统上工作的所有用户提供一个安全、有效的环境。从系统引导直至系统关闭，shell 脚本一直在运行。它们被用来进行系统初始化、监视进程、安装软件、检测磁盘使用情况、调度任务等。这样，除了系统日常例行工作之外，系统管理员的大部分时间用在管理用户账户上，包括从登录 shell 到运行初始化脚本、在 shell 中执行命令以至最后退出 shell 的所有工作。本书帮助各种类型的 UNIX/Linux 用户了解了主流 shell 的工作原理以及如何读写脚本。本章介绍了一些适合于系统管理的 shell 脚本的要点以及它们是如何与系统、用户进行交互的。有些主题在这里看起来是不相关的，但从整个系统管理的角度来讲却是十分重要的。下面列出了一些有用的资源。

Nemeth，E.，Snyder，G.，Seebass，S.，Hein，T. R.，*UNIX System Administration Handbook*，3rd Ed.，Upper Saddle River，NJ: Prentice Hall PTR，2000. ISBN: 0131510517。

Nemeth，E.，Snyder，G.，Hein，T. R.，*Linux Administration Handbook*，*Upper Saddle River*，NJ: Prentice Hall PTR，2002. ISBN: 0130084662。

Gagné，M.，*Linux System Administration: A User's Guide*，Boston: Addison-Wesley，2001. ISBN: 0201719347。

Sobell，M. G.，*A Practical Guide to Red Hat Linux: Fedora Core and Red Hat Enterprise* Linux，2nd Ed.，Pearson Education，2004. ISBN: 0131470248。

也可以参考 www.ugu.com，Unix Guru Universe：UNIX 系统管理员的官方主页。



appendix



常用的 UNIX/Linux 实用程序

apropos—在 whatis 数据库中搜索字符串

apropos keyword ...

apropos 用于搜索包含有系统命令关键字简短描述的一组数据库文件(参见目录 /usr/man/whatis)，并将结果显示在标准输出上。同 man –k。

范例 A-1

```
1 $ apropos bash
  bash (1)                   - GNU Bourne-Again SHell

2 $ man -k tcsh
  tsh (1)            - C shell with filename completionand command-line editing
```

说明

1. apropos 搜索关键字 bash 并打印关于 bash 的简短描述。
2. man –k 与 apropos 行为相同。

arch—打印机器的体系结构(参见 uname -m)

arch

在当前的 Linux 系统上，arch 打印诸如 i386、i486、i586、alpha、sparc、arm、m68k、mips 或 ppc 等内容。

范例 A-2

```
$ arch
i386
```

at、batch – 在将来某时刻执行命令

```
at [-csm] [-f script] [-qqueue] time [date] [+ increment]
at -l [ job...]
at -r job...
batch
```



at 和 batch 从标准输入读取命令，在将来某一时刻执行它们。at 用来指定在何时执行命令，batch 则将作业排队，在系统负载允许时执行它们。目标命令来自标准输入或文件，被安排在将来某时刻执行。如果没有被重定向，输出结果将通过邮件发给用户。

范例 A-3

```
1   at 6:30am Dec 12 < program
2   at noon tomorrow < program
3   at 1945 pm August 9 < program
4   at now + 3 hours < program
5   at 8:30am Jan 4 < program
6   at -r 83883555320.a
```

说明

1. 于 12 月 12 日早上 6:30 启动作业。
2. 于明天中午启动作业。
3. 于 8 月 9 日晚 7:45 启动作业。
4. 于 3 小时后的时刻启动作业。
5. 于 1 月 4 日早上 8:30 启动作业。
6. 删除此前安排的作业 83883555320.a。

awk – 扫描模式并处理语言

```
awk [ -fprogram-file ] [ -Fc ] [ prog ] [ parameters ] [ filename...]
```

awk 扫描每个输入文件名指定的文件，以查找与 proc 指定的模式组中任一模式匹配的行。

范例 A-4

```
1   awk '{print $1, $2}' file
2   awk '/John/{print $3, $4}' file
3   awk -F: '{print $3}' /etc/passwd
4   date | awk '{print $6}'
```

说明

1. 打印文件 file 中各行的头两个字段，文件中的字段以空白符分隔。
2. 找到模式 found 后，打印所在行的第 3 个和第 4 个字段。
3. 把冒号作为字段分隔符，打印文件/etc/passwd 的第 3 个字段。
4. 把 date 命令的输出结果发给 awk，由 awk 打印出结果的第 6 个字段。

banner – 制作标语

banner 用大型号的字符把它的参数(每个参数最长为 10 个字符)打印在标准输出上。

范例 A-5

```
banner Happy Birthday
```

说明

用标语的形式显示字符串 Happy Birthday。

basename – 提取给定目录名中的部分路径名

```
basename string [ suffix ]
dirname string
```

basename 从字符串 string 中删除以正斜杠/结尾的前缀，以及字符串 string 中可能包含的指定后缀 suffix，然后将结果打印在标准输出上。

范例 A-6

```
1  basename /usr/local/bin
2  scriptname="`basename $0`"
```

说明

1. 删除前缀/usr/local/，结果为 bin。
2. 仅将脚本的名称，$0，赋值给变量 scriptname。

bash – GNU Bourne Again shell

```
bash [options] [file[arguments]]
sh [options] [file[arguments]]
```

bash 版权信息为 Copyright © 1989, 1991 by the Free Software Foundation,Inc。bash 是一个与 sh 兼容的命令语言解释器，它从标准输入或文件中读取并执行命令。bash 同时还结合了来自 Korn shell 和 C shell(ksh 和 csh)的有用的特性。

bc – 处理高精度的算术运算

```
bc [ -c ] [ -l ] [ filename...]
```

bc 是一个交互式的语言处理程序，处理类似于 C 语言的对象，但能提供高精度的算术运算。bc 先从所有指定文件中读取输入，处理完之后，接着从标准输入读取。

范例 A-7

```
1  bc << EOF
   scale=3
   4.5 + 5.6 / 3
   EOF
   Output : 6.366
   ------------------------------
2  bc
   ibase=2
   5
   101 (Output)
   20
   10100 (Output
   ^D
```

说明

1. 这是一个 here 文档。从第一个 EOF 到结尾处那个 EOF 之间的输入被传给 bc 命令。



scale 用于指定小数点后的位数。计算结果被显示在屏幕上。

2. 数基为 2。所给的整数被转换为二进制形式(只有 AT&T 版本提供这一功能)。

bdiff – 比较两个大文件

如果需要比较的文件过大，diff 不能处理，可使用 bdiff。

cal – 显示日历

```
cal [ [ month ] year ]
```

cal 可打印出指定年份的日历。如果还指定了月份，cal 就只打印该月的日历。如果年、月均未指定，则 cal 打印当前月的日历。

范例 A-8

```
1  cal 1997
2  cal 5 1978
```

说明

1. 打印 1997 年的日历。
2. 打印 1978 年 5 月的日历。

cat – 连接文件并显示

```
cat [ -bnsuvet ] filename...
```

cat 按顺序读入指定的每个文件，把它们的内容写到标准输出。如果未指定输入文件，或指定了参数-，cat 就从标准输入文件读取输入。

范例 A-9

```
1  cat /etc/passwd
2  cat -n file1 file2 >> file3
```

说明

1. 显示文件/etc/passwd 的内容。
2. 连接文件 file1 和 file2，并将结果追加到文件 file3 中。开关-n 使得每一行都被标上行号。

chfn – 改变 finger 信息

```
chfn  [ -f full-name ] [ -o office ] [ -poffice-phone ]
      [ -h home-phone ] [ -u ] [ -v ] [ username ]
```

chfn 用于改变 finger 信息。该信息存储在/etc/passwd 文件中，由 finger 程序显示。Linux finger 命令显示可由 chfn 改变的 4 部分信息：真实姓名、工作单位、工作电话和家庭电话。

chmod – 改变文件的权限模式

```
chmod [ -fR ] mode filename...
```

```
chmod [ugoa ]{ + | - | = }[ rwxlsStTugo] filename...
```

chmod 用于修改或设置文件的模式。文件的模式规定了文件的访问权限等属性。模式可以是绝对的或链接的。

范例 A-10

```
1   chmod +x script.file
2   chmod u+x,g-x file
3   chmod 755 *
```

说明

1. 打开属主、属组和其他用户对文件 script.file 的执行权限。
2. 打开属主对文件 file 的执行权限，同时取消属组用户的执行权限。
3. 为当前工作路径下所有文件打开属主用户的读、写和执行权限、属组用户的读和执行权限，以及其他用户的读和执行权限。所给的参数是一个八进制值(111 101 101)，相当于：
   rwxxr-xr-x

chown – 改变文件的所有者

```
chown [ -fhR ] owner filename ...
```

chown 把文件的所有者改为 owner。owner 可以是十进制的用户 ID，也可以是文件/etc/passwd 中的登录名。只有文件的属主(或超级用户才能)改变该文件的属主。

范例 A-11

```
1   chown john filex
2   chown -R ellie ellie
```

说明

1. 把文件 filex 的用户 ID 改为 john。
2. 逐级扫描目录，将目录 ellie 下所有文件的属主改为 ellie。

chsh – 改换登录 shell

```
chsh [ -s shell ] [ -l ] [ -u ] [ -v ] [ username ]
```

chsh 用于改换登录 shell。如果命令行中没有给出 shell，则 chsh 将提示用户进行选择。所有有效的 shell 列于文件/etc/shells 中：

-s，--shell	指定登录 shell
-l，--list-shells	打印列于文件/etc/shells 中的 shell 并退出
-u，--help	打印使用信息并退出
-v，--version	打印版本信息并退出

范例 A-12

```
1 $ chsh -l
/bin/bash
/bin/sh
```




```
/bin/ash
/bin/bsh
/bin/tcsh
/bin/csh
/bin/ksh
/bin/zsh

2 $ chsh
Changing shell for ellie.
New shell [/bin/sh] tcsh
chsh: shell ust be a full pathname.
```

说明

1. 列出该 Linux 系统上所有可用的 shell。

2. 请求用户键入新的登录 shell 的完全路径名。除非给出类似/bin/tcsh 之类的完全路径名，否则该命令会失败。

clear – 清空终端屏幕

cmp – 比较两个文件

```
cmp [ -l ] [ -s ] filename1 filename2
```

cmp 将比较文件名已知的两个文件，如果二者相同，cmp 不做任何表示。如果二者不同，cmp 就指出从哪一行的哪个字节开始不同。

范例 A-13

```
cmp file.new file.old
```

说明

如果这两个文件有差异，就显示出差异从哪一行的哪个字符开始。

compress – 压缩、解压、zcat 压缩、zcat 解压文件，或显示展开的文件

```
compress [ -cfv ] [ -b bits ] [ filename... ]
uncompress [ -cv ] [ filename... ]
zcat [ filename... ]
```

compress 采用自适应的 Lempel-Ziv 编码来缩小指定文件的长度。compress 尽可能将文件名加上扩展名.Z，文件的所有者、访问时间和修改时间则不变。如果没有指定文件，compress 就将标准输入的内容压缩到标准输出。

范例 A-14

```
1   compress -v book
    book:Compression:35.07% -- replaced with book.Z
2   ls
    book.Z
```

说明

把文件 book 压缩为 book.Z，并显示出文件的压缩比和新名称。

cp – 复制文件

```
cp [ -i ] [ -p ] [ -r ] [ filename ... ] target
```

cp 命令将文件 filename 复制到 target，target 可以是一个文件或目录。filename 和 target 这两个参数不能相同。如果 target 不是目录，target 前面就只能指定一个文件；如果 target 是个目录，就可以指定多个文件。如果 target 不存在，cp 就创建一个名为 target 的文件。如果 target 存在并且不是目录，它的内容就会被覆盖。如果 target 是一个目录，指定的文件就被复制到该目录下。

范例 A-15

```
1  cp file1 file2
2  cp chapter1 book
3  cp -r desktop /usr/bin/tester
```

说明

1. 把文件 file1 的内容复制到文件 file2。
2. 把文件 chapter1 的内容复制到目录 book 下。复制到目录 book 的文件保留了原来的名字 chapter1。
3. 逐层将整个 desktop 目录复制到/usr/bin/tester。

cpio – 从/往文件档案复制内容

```
cpio -i [ bBcdfkmrsStuvV6 ] [ -C bufsize ] [ -E filename ]
  [ -H header ] [ -I filename [ -M message ] ] [ -R id ]   [ pattern ... ]
cpio -o [ aABcLvV ] [ -C bufsize ] [ -H header ]
  [ -O filename [ -M message ] ]
cpio -p [ adlLmuvV ] [ -R id ] directory
```

cpio 按照所给的修饰符复制归档文件，通常用于将文件备份到磁带或目录。

范例 A-16

```
find . -depth -print | cpio -pdmv /home/john/tmp
```

说明

find 从当前目录开始，逐级向下遍历目录结构，打印出每个目录的目录项，无论是否对该目录有写权限。这些文件名都被发给 cpio，cpio 将对应的文件复制到/home 分区下的 john/tmp 目录。

cron – 时钟监护进程

cron 在指定的日期和时间执行命令。可以在文件/etc/crontab 中指定需要定期执行的任务。但必须满足下列条件之一才能使用 cron：(1)超级用户；(2)普通用户，但用户 ID 被列在文件/etc/cron.allow 中；(3)普通用户，但系统中存在一个内容为空的/etc/cron.deny 文件。



crypt – 加密或解密文件

```
crypt [ password ]
```

crypt 对文件内容进行加密和解密。password 是密钥，它决定了转换的类型。

cut – 删除文件各行中的指定字段或字符

```
cut -clist [ filename ... ]
cut -flist [ -dc ] [ -s ] [ filename ... ]
```

cut 命令从文件的各行中截除某一(或某几)列或字符；如果未指定文件，cut 便对标准输入进行操作。选项-d 指定字段分隔符。默认的字段分隔符是制表符。

范例 A-17

```
1、 cut -d: -f1,3 /etc/passwd
2   cut -d: -f1-5 /etc/passwd
3   cut -c1-3,8-12 /etc/passwd
4   date | cut -c1-3
```

说明

1. 以冒号作为字段分隔符，显示文件/etc/passwd 的第 1、3 字段。
2. 以冒号作为字段分隔符，显示文件/etc/passwd 的第 1~5 字段。
3. 截除文件/etc/passwd 中各行的第 1~3、8~12 个字符，并显示在屏幕上。
4. 将 date 命令的输出结果作为输入传给 cut。打印结果的头 3 个字符。

date – 显示日期和时间或设置日期

```
[ -u ] [ -a [ - ] sss.fff ] [ yymmddhhmm [ .ss ] ] [+format ]
```

不带参数时，date 命令显示日期和时间。如果命令行参数以加号开头，则参数的剩余部分被用于格式化输出结果。百分号后面的字符是格式字符，用于提取日期中特定部分，比如年份或星期。设置日期时，命令行参数应用数字来表达，分别代表年、月、日、时和分。

范例 A-18

```
1   date +%T
2   date +20%y
3   date "+It is now %m/%d /%y"
```

说明

1. 将时间显示为 20:25:51。
2. 显示 2096。
3. 显示 “It is now 07/25/96.”

dd – 复制文件的同时进行转换

```
dd [--help] [--version] [if=file] [of=file][ibs=bytes] [obs=bytes]
```

```
[bs=bytes] [cbs=bytes] [skip=blocks] [seek=blocks] [count=blocks]
[conv={ascii,ebcdic,ibm,block,unblock,lcase,ucase,swab,noerror,notrunc,
sync}]
```

将文件从一个位置复制到另一个位置，通常是复制到磁带或从磁带中复制，或者在不同的操作系统之间进行复制。

范例 A-19

```
1   $ dd --help
2   $ dd if=inputfile of=outputfile conv=ucase
```

说明

1. 打印所有的选项和标识，并一一附上简短的说明。
2. 将 inputfile 文件中所有的字符转换为大写并将输出发送到 outputfile 文件。

diff – 比较两个文件的不同

```
[-bitw] [-c | -Cn]
```

比较两个文件，按行显示二者的差别。同时显示出相应的命令，您可以在 ed 编辑器中用它们实现这些差别。

范例 A-20

```
diff file1 file2
1c1
< hello there
---
> Hello there.
2a3
> I'm fine.
```

说明

显示文件 file1 和 file2 中对应各行有什么不同。符号<代表第 1 个文件，符号>则代表第 2 个文件。每一行的比较结果前面都有一条 ed 编辑命令，用这条编辑命令就能让两个文件的对应部分变得一样。

dos，xdos，dosexec，dosdebug – 在 linux 系统上运行 MS-DOS 和 MS-DOS 程序的 Linux DOS 仿真器

参见 Linux 帮助中的详细描述，这里不再赘述。

df – 统计剩余磁盘空间

```
df [-aikPv] [-t  fstype] [-x fstype] [--all][--inodes][--type=fstype]
   [--exclude-type=fstype][--kilobytes] [--portability] [--print-type]
   [--help] [--version] [filename...]
```

df 命令显示的是驻留单个文件的文件系统信息，默认则显示所有的文件系统。



范例 A-21

```
df
Filesystem        1024-blocks  Used Available Capacity
Mounted on
/dev/hda5           1787100 1115587   579141     66%   /
```

du – 磁盘使用概况

```
du [-arskod] [name ...]
```

du 命令将显示指定文件或指定目录下所有文件和目录(包括各级子目录)占用的磁盘块数量，每 512 字节为一块。

范例 A-22

```
1   du -s /desktop
2   du -a
```

说明

1. 显示/desktop 及其子目录下所有文件占用的磁盘块总数。
2. 显示当前目录及其子目录下各文件占用的磁盘块数目。

echo – 回显参数

```
echo [ argument ] ...
echo [ -n ] [ argument ]
```

echo 命令把传给它的参数写到标准输出上，用空格分隔各参数，并且在输出的末尾加上一个换行符。

针对 System V 系统的 echo 命令选项：

\b 退格

\c 取消输出结果末尾的换行符

\f 换页

\n 换行

\r 回车

\t 制表符

\v 纵向制表符

\\ 反斜杠

\0n n 为 1、2 或 3，八进制值

egrep – 在文件中查找模式，使用完整的正则表达式

```
egrep [ -bchilnsv ] [ -e special-expression ][ -f filename ]
[ strings ] [ filename ... ]
```

egrep(expression grep，表达式 grep)在文件中查找字符模式，打印出包含该模式的所有行。egrep 使用完整的正则表达式来匹配模式，所谓完整的正则表达式，是指这些表达式中

的字符串值使用的是完整的字符数字字符和特殊字符集。

范例 A-23

```
1   egrep 'Tom|John' datafile
2   egrep '^ [A-Z]+' file
```

说明

1. 显示文件 datafile 中包含模式 Tom 或 John 的所有行。
2. 显示以一个或多个大写字母开头的所有行。

expr – 将参数作为表达式进行计算

expr 参数

参数被视为表达式。expr 计算完这些表达式后，把结果写到标准输出。表达式中各项间必须由空格分隔。对 shell 有特殊含义的字符必须被转义。Bourn shell 脚本用 expr 来执行简单的算术运算。

范例 A-24

```
1   expr 5 + 4
2   expr 5 \* 3
3   num=0
    num=`expr $num + 1`
```

说明

1. 打印 5 + 4 的和。
2. 打印 5 * 3 的结果。用反斜杠保护星号不被 shell 扩展。
3. 将变量 num 赋值为 0 后，expr 命令将 num 加 1，并且将结果赋值给 num。

fgrep – 在文件中查找字符串

```
fgrep [ -bchilnsvx ] [ -e special string ]
[ -f filename ] [ strings ] [ filename ... ]
```

fgrep(快速 grep)在文件中查找字符串，并且打印出包含该字符串的所有行。fgrep 与 grep 和 egrep 的区别在于它把正则表达式元字符解释为字面字符。

范例 A-25

```
1   fgrep '***' *
2   fgrep '[ ] * ? $' filex
```

说明

1. 显示当前目录下所有文件中包含 3 个星号的所有行。所有字符都被当成其自身，即元字符也不会被特殊处理。

2. 显示文件 filex 中包含引号括着的那个字符串的所有行。

```
file [[ -f ffile ] [ -cl ] [ -m mfile ] filename...
```



file – 通过查看文件内容来确定其类型

file 对命令行指定的每个文件执行一系列测试，以确定文件包含的内容。如果文件的内容像是 ASCII 文本，file 就检查它的头 512 个字节，然后试着猜出该文件所用的语言。

范例 A-26

```
1   file bin/ls
    /bin/ls: sparc pure dynamically linked executable
2   file go
    go: executable shell script
3   file junk
    junk:  English text
```

说明

1. ls 是二进制文件，在执行时被动态链接。
2. go 是可执行的 shell 脚本。
3. junk 是包含 ASCII 文本的文件。

find – 查找文件

```
find path-name-list expression
```

find 在路径名列表 pathname list 指定的每个路径下(包含各级子目录)查找符合选项的文件。第一个参数是一个路径，指定查找从哪开始。其余的参数则规定目标文件需要满足的条件，如文件的名称、长度、属主、权限等。不同版本的 UNIX 上的 find 的语法也不同，请查看 UNIX 的手册页。

范例 A-27

```
1   find . -name \*.c -print
2   find .. -type f -print
3   find . -type d -print
4   find / -size 0 - exec rm "{}" \;
5   find ~ -perm 644 -print
6   find . -type f -size +500c -atime +21 -ok rm -f "{}" \;
7   find . -name core -print 2> /dev/null   (Bourne and Korn Shells)
    ( find . -name core -print > /dev/tty ) >& /dev/null   (C shell)
8   find / -user ellie xdev -print
9   find ~ -atime +31 -exec mv {} /old/{} \; -print
```

说明

1. 从当前工作目录开始(用句点代表)，查找所有名字以.c 结尾的文件，并且打印所找到文件的完整路径名。

2. 从当前工作目录的父目录开始(用两个句点代表)，查找所有类型为 file 的文件，即所有不是目录的文件。

3. 从当前工作目录开始(用句点代表)，查找所有目录文件。

4. 从根目录开始，查找所有长度为 0 的文件并删除它们。{}用来代表找到的各个文件的名称。

5. 从该用户的主目录(Korn 和 C shell 用~代表当前用户的主目录)开始，查找所有权限为 644(属主用户可读写，属组用户和其他用户只能读)的文件。

6. 从当前工作目录开始，查找长度超过 500 个字节、在最近 21 天内未被访问过的文件，并且询问是否需要删除它们。

7. 从当前工作目录开始，查找并显示所有名称为 core 的文件，同时将报错信息发送到 /dev/null，/dev/null 是 UNIX 的位桶。

8. 打印出 root 分区上属于用户 ellie 的所有文件的名称。

9. 把已有超过 31 天未被访问的文件移到目录/old 下，同时打印出这些文件的名称。

finger – 显示本地用户和远程用户的相关信息

```
finger [ -bfhilmpqsw ] [ username... ]
finger [-l] username@hostname...
```

默认情况下，finger 命令显示每个已登录用户的相关信息，包括他的登录名、全名、终端名(没有写权限的终端的名字前有一个星号)、空闲时间、登录时间和所在位置(如果知道的话)。

fmt – 简单的文本格式处理程序

```
fmt [ -c ] [ -s ] [ -w width | -width ] [ inputfile... ]
```

fmt 是一个简单的文本格式处理程序，它能填充或连接文本行，使输出的文本行包含(不超过)宽度选项-w 指定的字符数。默认宽度为 72。fmt 会把参数中列出的输入文件的内容连接起来。如果未指定输入文件，fmt 就格式化来自标准输入的文本。

范例 A-28

```
fmt -c -w45 letter
```

说明

fmt 命令将格式化文件 letter。开关-c 让 fmt 保留每段中头两行的缩进，后面的各行设置为与第二行相同的左边距。开关-w 指示 fmt 填充输出行，使其包含 45 列。

fold – 折叠长的文本行

```
fold [ -w width | -width ] [ filename ... ]
```

fold 命令折叠指定文件的内容，断开文件中的文本行，使其不超过最大宽度。如果未指定文件，fold 就对标准输入进行操作。文本行的默认宽度是 80。包含制表符时，文本行宽度必须是 8 的倍数，否则就要将制表符展开。

ftp – 文件传输程序

```
ftp [ -dgintv ] [ hostname ]
```

ftp 命令是用户与 Internet 上标准的文件传输协议(FTP)的接口。ftp 将从/往远程网络站点传输文件。不只是 UNIX 机器，其他操作系统的机器上也有文件传输程序。



范例 A-29

```
1   ftp ftp.uu.net
2   ftp -n 127.150.28.56
```

说明

1. ftp 到机器 ftp.uu.net，这是由 UUNET 服务管理的一个大型资料库，它为 UNIX 系统处理电子邮件和网络新闻。

2. 打开到 127.45.4.1 这台机器的连接，但不自动登录。

free – 显示系统空闲和已用内存数量

```
free [-b | -k | -m] [-o] [-s delay ] [-t] [-V]
```

free 显示的是系统上空闲和已用物理内存、交换内存以及内核使用的共享内存和缓冲的总数。

范例 A-30

```
% free
          total     used      free      shared   buffers   cached
Mem:      64148     54528     9620      45632    3460      29056
-/+ buffers/cache:  22012     42136
Swap:     96352     0         96352
```

fuser – 识别使用文件或套接字的进程

```
fuser [-a|-s] [-n space] [-signal] [-kmuv] name ...[-] [-n space]
  [-signal] [-kmuv] name ...
fuser -l
fuser -V
```

fuser 命令用于显示使用了指定文件或文件系统的进程的 PID。默认显示模式在每个文件名后跟一个代表该文件访问类型的字母。

范例 A-31

```
% fuser --help
usage: fuser [ -a | -q ] [ -n space ] [ -signal ] [ -kmuv ]
filename ... [ - ] [ -n space ] [ -signal ] [ -kmuv ] name
...
       fuser -l
       fuser -V
    -a       display unused files too
    -k       kill processes accessing that file
    -l       list signal names
    -m       mounted FS
    -n       space  search in the specified name space (file, udp, or tcp)
    -s       silent operation
    -signal   send signal instead of SIGKILL
    -u       display user ids
    -v       verbose output
    -V       display version information
    -        reset options
```

```
udp/tcp names: [local_port][,[rmt_host][,[rmt_port]]]
```

gawk – 模式扫描与语言处理

```
gawk [ POSIX or GNU style options ] -f program-file [-- ] file ...
gawk [ POSIX or GNU style options ] [ -- ]program-text file ...
```

gawk 是 awk 编程语言的 GNU 工程实现。它遵循 POSIX 1003.2 命令语言与工具标准中关于该语言的定义。该版本依次建立在 Aho，Kernighan 和 Weinberger 在 *The AWK Programming Language* 中描述的基础上。另外还增加了 System V Release 4 版本的 UNIX awk 的一些特性。gawk 最近也提供了贝尔实验室 awk 扩展和一些 GNU 规范扩展。

gcc，g++ – GNU 项目的 C 与 C++语言编译器(V2.7)

```
gcc [ option | filename ]...
g++ [ option | filename ]...
```

getopt(s) – 解析命令行选项

getopts 命令取代了 getopt。getopts 用于拆分命令行中的选项，方便 shell 过程对参数的解析。getopts 还会检查选项的合法性。

grep – 在文件中查找模式

```
grep [ -bchilnsvw ] limited-regular-expression [ filename ... ]
```

grep 在文件中查找指定模式，并且打印出包含该模式的所有行。grep 使用正则表达式元字符来匹配模式。egrep 使用的则是扩展的元字符集。

范例 A-32

```
1  grep Tom file1 file2 file3
2  grep -in '^tom savage' *
```

说明

1. grep 显示文件 file1、file2 和 file3 中所有包含模式 Tom 的行。

2. grep 显示当前工作目录下的文件中包含 tom savage 的所有行，并且给出各自的行号。这条 grep 命令忽略大小写，但要求 tom savage 必须出现在行首。

group – 打印用户的组隶属关系

```
groups [ user... ]
```

groups 命令在标准输出上打印出当前用户或某个指定用户所属的全部组。

gzip，gunzip，zcat – 压缩或解压缩文件

```
gzip [ -acdfhlLnNrtvV19 ] [-S suffix] [ name ... ]
gunzip [ -acfhlLnNrtvV ] [-S suffix] [ name ... ]
zcat [ -fhLV ] [ name ... ]
```



gzip 使用 Lempel-Ziv 编码算法(LZ77)减少指定文件的大小。只要可能，每个文件都被带扩展名.GZ 的文件代替，同时保持原来的所有权模式，访问权限和修改时间。

head – 输出文件的前 10 行

```
head [-c N[bkm]] [-n N] [-qv] [--bytes=N[bkm]] [--lines=N] [--quiet]
  [--silent] [--verbose] [--help] [--version] [file...]
head [-Nbcklmqv] [file...]
```

head 在标准输出上显示文件的前十行。多于一个文件时，在每行前面加上文件名。如果没有文件或文件为-，则从标准输入读。

host – 显示指定 DNS 中的主机或域的信息

```
host [l] [-v] [-w] [-r] [-d] [-t querytype] [a] host [server]
```

host 命令显示指定互联网主机的信息。它从跨国互联的一组服务器上获取信息。默认情况下，它在主机名和 IP 地址间进行转换，使用-a 或-t 开关，所有信息都被显示出来。

id – 显示当前用户的用户名、用户 ID、所属组名和组 ID

```
/usr/bin/id [ -a ]
```

id 显示当前用户的用户 ID、用户名、组 ID 和组名。如果用户的真实 ID 和有效 ID 不相同，就两个都打印。

jsh – 标准的作业控制 shell

```
jsh [ -acefhiknprstuvx ] [ argument...]
```

jsh 命令是对应标准 Bourne shell 的接口，它能提供 Bourne shell 的所有功能，并且允许作业控制。

kill – 发送一个信号以终止一个或多个进程

```
kill [
```

kill 发送一个信号以终止一个或多个进程 ID。

killall – 终止指定名称的进程

less – 与 more 相对的命令

```
less -?
less --help
less -V
less --version
less [-[+]aBcCdeEfgGiImMnNqQrsSuUVwX][-b bufs] [-h lines] [-j line]
  [-k keyfile][-{oO} logfile] [-p pattern] [-P prompt] [-t tag]
  [-T tagsfile] [-x tab] [-y lines] [-[z] lines] [+[+]cmd] [--]
  [filename]...
```

less 是一个与 more 类似的程序，但是它能够在文件中向前和向后移动。另外，less 并不是读完了整个输入文件之后才开始显示，因此对较大的输入文件它比其他文本编辑器如 vi 启动要快很多。less 使用 termcal(有些系统上使用 terminfo)，因此它可以在各种终端上运行。它甚至对硬拷贝终端(hardcopy terminal)也能提供一定的支持。

line – 读取一行

line 从标准输入复制一行(到换行符为止)，然后写到标准输出上。遇到 EOF 时，line 返回退出值 1。任何情况下，line 至少会打印出一个换行符。这条命令通常是在 shell 文件中用于从用户终端读取输入。

ln – 为文件创建硬链接

```
ln [options] source [dest]
ln [options] source... directory
Options:
[-bdfinsvF] [-S backup-suffix] [-V {numbered,existing,simple}]
[--version-control={numbered,existing,simple}] [--backup]
[--directory] [--force][--interactive] [--no-dereference] [--symbolic]
[--verbose] [--suffix=backup-suffix] [--help] [--version]
```

如果最后一个参数是已经存在的某个目录，则 ln 将其他给定的文件链接到该目录中的同名文件。如果仅给出了一个文件，则将该文件链接到当前目录。另外，如果仅给出了两个文件，则将第一个文件链接到第二个文件。如果最后一个参数不是目录且给出的文件多于两个，则报错。当跨分区时必须使用符号链接。

选项如下所示。

-b，--backup	备份即将被删除的文件
-d，-F，--directory	允许超级用户对目录建立硬链接
-f，--force	删除已存在的目标文件
i，--interactive	提示用户是否确认删除已存在的目标文件
-n，--no-dereference	当指定的目标文件是某个目录的符号链接时，尝试在目录中创建一个它指向的链接以取代原符号链接而非撤消原符号链接
-s，--symbolic	创建符号链接而非硬链接
-v，--verbose	建立链接前显示每个文件的名称
--help	在标准输出上显示用法信息，并成功退出
--version	在标准输出上显示版本信息，并成功退出
-S，--suffix backup-suffix	在备份文件后添加后缀，如果不使用该选项则使用环境变量 SIMPLE_BACKUP_SUFFIX。二者是等价的。环境变量可以设置该选项，该选项也可以改写此环境变量。如果没有使用选项，同时环境变量也未被设置，则就如同 emacs 一样使用默认值~
-V, --version-control {numbered，existing，simple}	可以使用环境变量 VERSION_CONTROL 设置备份模式类型



范例 A-33

```
1   ls -l
    total 2
    drwxrwsr-x  2 ellie   root        1024 Jan 19 18:34 dir
    -rw-rw-r--  1 ellie   root        16   Jan 19 18:34 filex
2   % ln filex dir
3   % cd dir
4   % ls -l
    total 1
    -rw-rw-r--  2 ellie   root          16 Jan 19 18:34 filex
```

说明

1. ls 命令的输出显示了目录 dir 和文件 filex 的一个长列表。目录的链接数总是至少为 2，一个是目录自身，另一个由父目录使用。文件的链接数总是至少为 1，将它链接到创建该文件的目录。文件被删除后，它的链接数降为 0。

2. ln 命令创建一个硬链接。filex 此时链接到目录 dir 和当前目录上。链接并不创建新文件。它只是简单地给已存在的文件增加一个名字或给已存在的目录增加一种查找路径。如果删除其中一个链接，另一个依然存在。对被链接文件的任何修改都将影响到其他的链接，因为它们实际上是同一个文件。

3. 将目录切换到 filex 链接的目录上。

4. filex 的链接数为 2。同样一个文件，但是既可以从当前目录访问，也可以从父目录进行访问。

logname – 获取运行进程的用户的名字

look – 显示以给定字符串开头的行

```
look [-dfa] [-t termchar] string [file]
```

look 显示文件中以某个字符串作为前缀的行。因为 look 执行的是二进制搜索，所以文件中的行必须被排序。如果文件未指定，则使用文件/usr/dict/words，仅比较字母数字字符，并且字母的大小写被忽略。

选项如下所示。

-d	使用字典字符集和字典序。也就是说，仅比较字母数字字符
-f	忽略字母字符的大小写
-a	使用备用字典/usr/dict/web2
-t	指定一个字符串终止字符。也就是说，比较字符串中的字符直到遇到终止字符

如果找到一行或多行并显示出来，则 look 工具以 0 退出，如果没有找到任何行，则以 1 退出，如果出错，则以大于 1 的值退出。

范例 A-34

```
1   % look sunb
    sunbeam
```

```
    sunbeams
    Sunbelt
    sunbonnet
    sunburn
    sunburnt
2   % look karen sorted.datebook
3   % look Karen sorted.datebook
    Karen Evich:284-758-2857:23 Edgecliff Place, Lincoln, NB
    92086:7/25/53:85100
    Karen Evich:284-758-2867:23 Edgecliff Place, Lincoln, NB
    92743:11/3/35:58200
    Karen Evich:284-758-2867:23 Edgecliff Place, Lincoln, NB
    92743:11/3/35:58200
4   % look -f karen sorted.datebook
    Karen Evich:284-758-2857:23 Edgecliff Place, Lincoln, NB
    92086:7/25/53:85100
    Karen Evich:284-758-2867:23 Edgecliff Place, Lincoln, NB
    92743:11/3/35:58200
    Karen Evich:284-758-2867:23 Edgecliff Place, Lincoln, NB
    92743:11/3/35:58200
```

说明

1. 假定/usr/dict/words 在当前目录中，则 look 显示文件/usr/dict/word 中所有以字符串 sunb 开头的行。

2 .look 在文件 sorted.datebook 中未能找到以 karen 开头的行(该文件必须首先被排序，否则 look 将什么也找不到)。

3. look 显示了文件 sorted.datebook 中所有以字符串 Karen 开头的行。

4. -f 选项将忽略字符串中的大小写(也就是说，不区分大小写)。

lp – 将输出发往打印机(AT&T)

```
lp [ -cmsw ] [ -ddest ] [ -number ] [ -ooption ] [ -ttitle ] filename ...
cancel [ ids ] [ printers ]
```

lp 向行式打印机发送请求，cancel 取消请求。

范例 A-35

```
1   lp -n5 filea fileb
2   lp -dShakespeare filex
```

说明

1. 将文件 filea 和 fileb 各发 5 份到打印机。

2. 指定在打印机 Shakespeare 上打印 filex。

lpr -将输出发往打印机(UCB)

```
lpr [ -Pprinter ] [ -#copies ] [ -Cclass ] [ -Jjob ]
  [ -Ttitle ] [ -i [ indent ] ] [ -1234font ] [ -wcols ]
  [ -r ] [ -m ] [ -h ] [ -s ] [ -filter-option ] [ filename ... ]
```



lpr在假脱机池中创建一个打印机作业，等到设备空闲时进行打印。每个打印作业均由一个打印作业和一个或多个数据文件组成。

范例A-36

```
1  lpr -#5 filea fileb
2  lpr -PShakespeare filex
```

说明

1. 将文件filea和fileb各发5份到打印机。
2. 指定在打印机Shakespeare上打印filex。

lpstat – 打印LP打印服务的状态信息(AT&T)

lpq – 输出打印机的状态信息(UCB)

ls – 列出目录内容

```
ls [ -abcCdfFgilLmnopqrRstuxl ] [ names ]
```

对于每个目录参数，ls列出该目录的内容；对于每个文件参数，ls打印该文件的名称和所需的其他信息。默认情况下，输出结果按字母顺序排序。如果未给定参数，ls列出当前目录的内容。

范例A-37

```
1  ls -alF
2  ls -d a*
3  ls -i
```

说明

1. –a列出隐藏文件(即那些名称以句点开头的文件)。-l以长格式列出文件的属性。-F在目录文件名末尾加上斜杠，在可执行脚本名末尾加上星号(*)，在符号链接文件的名字后面加上符号@。
2. 如果开关-d的参数是某个目录，则只显示该目录的名称，而不显示内容。
3. 开关-i指示ls在每个文件名前面给出该文件的i节点号。

mail，rmail – 读邮件或向用户发送邮件

```
Sending mail
   mail [ -tw ] [ -m message_type ] recipient...
   rmail [ -tw ] [ -m message_type ] recipient...
Reading mail
   mail [ -ehpPqr ] [ -f filename ]
Forwarding mail
   mail -F recipient...
Debugging
   mail [ -x debug_level ] [ other_mail_options ] recipient...
   mail [ -T mailsurr_file ] recipient...
```

收件人是 login 可识别的用户名。如果指定了收件人，mail 就认为正在发送邮件。mail 可以从标准输入读取输入直至遇到文件结束符(或按下 Ctrl+D 组合键)，也可以从终端读取输入，直至遇到仅含一个句点的行为止。只要收到这些指示信号之一，mail 便将这封信添加到每个收件人的邮件文件。

mailx – 交互式的消息处理系统

```
mailx [ -deHiInNUvV ] [ -f [ filename|+folder ]]
   [ -T filename ] [ -u user ] [ recipient... ]
mailx [ -dFinUv ] [ -h number ] [ -r address ][ -s subject ] recipient...
```

上面列出的这些邮件工具提供了一个交互式的界面，用于发送、接收和处理邮件消息。其中有些功能必须在安装了基本的网络工具后才能使用。收到的邮件保存在称为 mailbox 的文件中，阅读过的邮件则被转移到文件 mbox 中。

make – 维护、更新或重新生成一组相关程序和文件

```
make [ -f makefile ] ... [ -d ] [ -dd ] [ -D ]
[ -DD ] [ -e ] [ -i ] [ -k ] [ -n ] [ -p ] [ -P ]
[ -q ] [ -r ] [ -s ] [ -S ] [ -t ] [ target ... ]
[ macro=value ... ]
```

make 根据某个描述文件中列出的命令来更新文件，如果目标文件比它依赖的同名文件，make 就更新目标文件。

man – 在线手册的格式设计与显示

```
man [-acdfhktwW] [-m system] [-p string] [-C config_file] [-M path]
    [-P pager] [-S section_list] [section] name...
```

manpath– 决定用户帮助的搜索路径

```
man [-acdfhkKtwW] [-m system] [-p string][-C config_file][-M path]
   [-P pager] [-S section_list][section] name  ...
```

man 设计并显示帮助手册。现在的版本有环境变量 MANPATH 和(MAN)PAGER，因此可以定义个人帮助并选择合适的程序来打开帮助页面。如果指定了节，man 仅显示手册中该节的内容。

mesg – 允许或禁止接收 write 命令产生的消息

```
mesg [ -n ] [ -y ]
```

带参数-n 时，mesg 通过取消用户终端的其他用户写权限来禁止用 write 写消息。带参数-y 的 mesg 则恢复该权限。如果什么参数都不带，mesg 报告当前的权限状态，不做任何修改。

mkdir – 创建目录

```
mkdir [ -p ] dirname ...
```



more – 浏览或逐页查看文本文件

```
more [ -cdflrsuw ] [ -lines ] [ +linenumber ] [ +/pattern ] [ filename ... ]
page [ -cdflrsuw ] [ -lines ] [ +linenumber ] [ +/pattern ] [ filename ... ]
```

more 是一个过滤器，它在终端上显示文本文件的内容，每次一屏。通常，more 每显示一屏就暂停，同时在屏幕底端打印—More-。

mtools – 访问 UNIX 上 DOS 盘的工具

mtools 是一个公共的工具集，它使得 UNIX 系统能够对 MS-DOS 文件系统(典型地如软盘)上的文件进行读、写和移动等操作。尽可能地，每个程序都将试着模仿 MS-DOS 等价的操作。但是，它们不模仿不必要的和怪异的 DOS 行为。例如，它能够将子目录从一个子目录移动到另外的子目录中。在以下地址(以及它们的镜像)可以找到 mtools：

http://mtools.ltnb.lu/mtools-3.9.1.tar.gz

ftp://www.tux.org/pub/knaff/mtools/mtools-3.9.1.tar.gz

ftp://sunsite.unc.edu/pub/Linux/utils/disk-management/mtools-3.9.1.tar.gz

mv – 移动或更名文件

```
mv [ -f ] [ -i ] filename1 [ filename2 ...] target
```

mv 命令将源文件移动到目标文件。源文件和目标文件不能重名。如果 target 不是目录，它前面就只能指定一个文件。如果 target 是目录，就能指定多个文件。如果 target 不存在，mv 就创建一个名为 target 的文件。如果 target 已存在并且不是目录，它的内容将被覆盖。如果 target 是目录，mv 就将源文件(一个或多个)移到该目录下。

范例 A-38

```
1   mv file1 newname
2   mv -i test1 test2 train
```

说明

1. 将文件 file1 更名为 newname。如果 newname 已存在，其内容将被覆盖。
2. 将文件 test1 和 test2 移动到目录 train。开关-i 指定使用交互模式，即在移动文件前进行询问。

nawk – 模式扫描和处理语言

```
nawk [ -F re ] [ -v var=value ] [ 'prog' ] [ filename ... ]
nawk [ -F re ] [ -v var=value ] [ -f progfile ][ filename ... ]
```

nawk 扫描每个输入文件，查找与指定模式组中任何一个相匹配的行。必须用单引号(')将命令串括起来，保护其不被 shell 解释。Awk 程序由一组模式/动作语句组成，可以用这些语句从文件、管道或标准输入中过滤出特定信息。

newgrp – 登录到一个新的组

```
newgrp [-] [ group ]
```

newgrp 改变用户的真实和有效组 ID，从而将用户登录到一个新的组中。用户保持已登录状态，当前目录也不会改变。执行 newgrp 通常会用一个新 shell 替换掉当前 shell，即便命令以错误终止(比如指定的组不可识别)。

news – 打印新闻条目

```
news [ -a ] [ -n ] [ -s ] [ items ]
```

news 用于通知用户当前发生的事件。依照惯例，这些事件是用目录/var/news 下的文件进行描述的。如果不带参数调用，news 将打印出当前/var/news 下所有文件的内容，越近的越靠前。并且会在每个文件前面加上一个合适的标头。

nice – 以低优先级运行命令

```
nice [ -increment ] command [ arguments ]
```

/usr/bin/nice 用较低的 CPU 调度优先级来执行指定命令。调用 nice 的进程(通常是用户的 shell)必须属于分时调度类。nice 在分时调度类内执行命令。默认的增量是 10。如果你不是超级用户，增量就必须在 1~19 之间。nice 也是 csh 的内置命令。

nohup – 使命令不响应挂起和退出信号

```
/usr/bin/nohup command [ arguments ]
```

nohup 有 3 个不同的版本。在 C shell 里，nohup 是一个内置命令，在 Bourne shell 中则是可执行文件/usr/bin/nohup。Bourne shell 版的 nohup 执行命令，使其不受 HUP(挂起)信号和 TERM(终止)信号的影响。标准输出如果是终端，就会被重定向到文件 nohup.out。标准错误输出随标准输出一起被重定向。命令的优先级被加 5。从 shell 调用 nohup 时应该加上&，以防止它响应中断和来自下一用户的输入。

范例 A-39

```
nohup lookup &
```

说明

lookup 程序将在后台运行，而且即使用户退出系统，也会持续运行直至完成。lookup 产生的所有输出都被写到当前目录下一个叫做 nohup.out 的文件中。

od – 八进制显示

```
od [ -bcCDdFfOoSsvXx ] [ filename ] [ [ + ] offset [ . ] [ b ] ]
```

od 根据第一个参数，用一种或多种格式显示文件。第一个参数如果未指定，则默认的为-o；文件可以显示为八进制、ASCII 码制、十进制、十六进制等格式。



pack, pcat, unpack – 压缩和展开文件

```
pack [ - ] [ -f ] name ...
pcat name ...
unpack name ...
```

pack 压缩文件。只要有可能(并且有用)，pack 就将输入文件 name 替换为压缩文件 name.z，name.z 的访问模式、访问和修改日期以及属主都与 name 的相同。文本文件通常能被压缩到原长度的 60-75%。除了不能用作过滤器外，pcat 对压缩文件做的操作与 cat 对普通文件所做的一样。指定的文件被解压并写到标准输出上。因此，要查看压缩文件 name.z，可以用命令 pcat name.z，或者就是 pcat name.unpack 来展开用 pack 生成的文件。

passwd – 修改登录口令和口令属性

```
passwd [ name ]
passwd [ -d | -l ] [ -f ] [ -n min ] [ -w warn ][ -x max ] name
passwd -s [ -a ]
passwd -s [ name ]
```

passwd 命令修改用户登录名对应的口令或列出口令属性。特权用户还可以用 passwd 来设置或修改与任何一个登录名对应的口令和口令属性。

paste – 合并多个文件中的相同行或同一文件连续行

```
paste filename1 filename2...
paste -d list filename1 filename2...
paste -s [ -d list ] filename1 filename2...
```

paste 连接给定的 filename1、filename2 等输入文件中相应的行。它把每个文件视为表的一列或多列，将它们横向粘贴在一起。

范例 A-40

```
1  ls | paste - - -
2  paste -s -d"\t\n" testfile1 testfile2
3  paste file1 file2
```

说明

1. 分 3 列将文件列出，列之间用制表符连接。

2. 将两行合成一行，用制表符和换行符作为分隔符，即头两行用制表符连接，接下来的两行用换行符连接，再往下的两行又用制表符连接，以此类推。开关-s 使得文件 testfile1 中的行贴在前头，后面才是 testfile2 中的行。

3. 将文件 file1 里的行加到文件 file2 的行后。二者用制表符连接，看起来就像表的两列。

pcat – (参见 pack)

pine– 一个用于 Internet 新闻组和电子邮件的程序

```
pine [ options ] [ address, address ]
```

```
pinef [ options ] [ address, address ]
```

pine 是一个面向屏幕的消息处理工具。在默认配置下，pine 有意面向初学者提供了一个受限功能集合。但它同时也有可供添加的选项、“权威用户”和个性化特性。pinef 是 pine 的一个变体，它使用功能键而不是需要记忆的单字母命令。pine 的基本特性集包括：视图、保存、导出、删除、打印、回复和消息转发。

pg – 逐页显示文件

```
pg [ -number ] [ -p string ] [ -cefnrs ] [ +linenumber ]
  [ +/pattern/ ] [ filename ... ]
```

pg 命令是个过滤器，您能用它在终端上逐屏分页浏览文件。如果未指定文件名，或所给的文件名为-，pg 将从标准输入读取输入。每一屏内容后都有一个提示。如果用户键入回车，pg 就显示下一页。它允许你退回重新检看之前看过的内容(参见“more”)。

pr – 打印文件

```
pr [[-columns] [-wwidth] [-a]] [-eck] [-ick] [-drtfp]
   [+page] [-nck] [-ooffset] [-llength] [-sseparator]
    [-    hheader] [-F] [filename ...]
pr [[-m] [-wwidth]] [-eck] [-ick] [-drtfp] [+page] [-nck]
   [-ooffset] [-llength] [-sseparator] [-hheader] [-F]
   [filename1 filename2 ...]
```

pr 命令根据不同的格式选项格式化并打印文件的内容。默认情况下，pr 把列表发送到标准输出并分成多页，每页的页头上都会表明页码、最后一次修改文件的日期和时间以及文件名。如果没有指定选项，则默认文件格式是 66 行，其中包括 5 行页眉和 5 行页脚。

范例 A-41

```
pr -2dh "TITLE" file1 file2
```

说明

分两列双面打印文件 file1 和 file2，页头设为“TITLE”。

ping – 在远程系统可达并处于活跃状态时报告

```
ping [-dfnqrvR] [-c count] [-i wait] [-l preload] [-p pattern]
  [-s packetsize]
```

ping 向一个主机发送 ICMP ECHO_REQUEST 报文并等待响应以验证主机或网关是可达的并处于活跃状态。它用于检测出网络连通性问题。如果 ping 没有收到回复报文则它将以状态值 1 退出。在出错时以状态值 2 退出。其他情况下退出状态值为 0。这样通过使用退出状态值就能够了解主机是否为活跃状态。

该程序一般用于网络测试、测量和管理。因为它会增加网络负载，因此在常规操作或自动运行的脚本中最好不要使用。



ps – 报告进程状态

```
ps [ -acdefjl ] [ -g grplist ] [ -p proclist ]
   [ -s sidlist ] [ -t term ] [ -u uidlist ]
```

ps 打印关于活动进程的信息。如果不带选项，ps 就打印与控制终端相关的进程信息。输出结果只包含进程 ID、终端标识符、累计执行时间和命令名。否则，所显示的信息则由选项来控制。AT&T 和 Berkeley 类型的 UNIX 版本上的 ps 选项不太一样。

范例 A-42

```
1   ps -aux | grep '^linda'        # ucb
2   ps -ef | grep '^ *linda'       # at&t
```

说明

1. 打印所有正在运行的进程，用管道将输出结果发送到 grep 程序，只打印属于用户 linda 的那些进程，每行都是以 linda 开头(UCB 版本)。

2. 和第一个例子相同，只适用于 AT&T 版本。

pstree – 显示进程树

```
pstree [-a] [-c] [-h] [-l] [-n] [-p] [-u] [-G|-U][pid|user]
pstree -V
```

pstree 将正在运行的进程以树状显示。该树以 pid 为根，在 pid 被省略时使用 init 作为根。如果指定了用户名，所有的进程树以该用户拥有的进程为根。pstree 通过将相同的分支放入到方括号中并以重复数为前缀从而将它们合并，例如：

```
init-+-getty
     |-getty
     |-getty
     '-getty
```

变为

```
init---4*[getty]
```

pwd – 显示当前工作目录名

quota– 显示用户磁盘空间使用和限制信息

```
quota [ -guvv | q ]
quota [ -uvv | q ] user
quota [ -gvv | q ] group
```

quota 显示的是用户的磁盘使用情况及所受的限制。默认情况下，仅打印用户配额。

-g　打印用户所在组的配额

-u　一个可选的标识，等价于默认状态

-v　显示未分配存储空间的文件系统的配额

-q　打印一些简短的信息，仅包含文件系统中使用空间已超过配额的相关信息

rcp – 远程文件复制

```
rcp [ -p ] filename1 filename2
rcp [ -pr ] filename...directory
```

rcp 命令按下列格式在机器之间复制文件：

```
remotehostname:path
user@hostname:file
user@hostname.domainname:file
```

范例 A-43

```
1   rcp dolphin:filename /tmp/newfilename
2   rcp filename broncos:newfilename
```

说明

1. 从远程机器 dolphin 将 filename 复制到本机的/tmp/newfilename。
2. 从本机复制 filename 到远程机器 broncos，并将其命名成 newfilename。

rdate – 通过网络获取日期和时间

```
rdate [-p] [-s] [host...]
```

rdate 通过 TCP 获取另外一台使用 RFC868 中协议的机器的当前时间。使用-p 选项，rdate 打印从远程机器上获取到的时间。这是默认模式。使用-s 选项，rdate 使用从远程机器获取到的时间来设置本地时间。仅超级用户能够重设时间。每个系统的时间以 ctime(3)格式返回。

范例 A-44

```
1   rdate homebound atlantis
(输出)
[homebound]   Tue Jan 18 20:35:41 2000
[atlantis]    Tue Jan 18 20:36:19 2000
```

rgrep – 递归的高级 grep 程序

```
rgrep [ options] pattern [file] ......
```

与 grep 和 egrep 不同，rgrep 能够在目录中递归。在 UNIX 系统上执行这类搜索的传统方式是将 find 命令与 grep 命令结合使用。使用 rgrep 性能要好很多。参见 xargs 命令。

命令行选项如下所示。

-?	附加帮助(在某些系统上使用-?可以避免 shell 扩展)
-c	匹配计数
-h	高亮显示匹配(假定是 ANSI 兼容的终端)
-H	输出匹配，而非包含匹配的整行
-i	忽略大小写



(续表)

-l	仅列出文件名
-n	打印匹配的行号
-F	跟随链接
-r	递归扫描目录树
-N	不执行递归搜索
-R pat	与-r 类似，不同的是仅检查匹配 pat 的文件
-v	仅打印与指定模式不匹配的行
-x ext	仅检查带给定后缀 ext 的文件
-D	打印所有可能被搜索的目录。该选项仅用于调试。该选项不查找任何文件
-W len	有 len 个字符的行(不以换行符终结)

支持的正则表达式

.	匹配除换行符外的任意字符
\d	仅匹配数字
\e	匹配 ESC 字符
*	匹配零个或多个如符号前面所示的正则表达式
+	匹配一个或多个如符号前面所示的正则表达式
?	匹配零个或一个如符号前面所示的正则表达式
^	匹配行首
$	匹配行尾
[...]	匹配括号中的单个字符。例如，[-02468]匹配-或任意偶数，[-0-9a-z]匹配-或 0~9 之间的数字或 a 至 z 之间的字母
\{ ...\}	用于重复。例如，x\{9\}匹配 9 个 x 字符
\(...\)	用于后向引用。\(...\)中的模式被标记并保存。从正则表达式最左侧开始，最多允许 9 个标签。要恢复保存的模式时，使用\1, \2 ... \9
\2 \1, =, ..., \9	匹配由第 n 个\(... \)表达式指定的模式。例如，\([\t][a-zA-Z]+\)\1[\t]匹配任何连续重复的词

范例 A-45

```
1  rgrep -n -R '*.c' '^int'
2  rgrep -n -xc '^int'
```

说明

1. 在当前目录及其子目录中所有以“c”为扩展名的文件中搜索行首为“int”的模式，带行号显示出所匹配的行。

2. 在所有以“c”为扩展名的文件中搜索行首为“int”的模式，带行号显示出所匹配的行(与上面一致)。

rlogin – 远程登录

```
rlogin [ -L ] [ -8 ] [ -ec ] [ -l username ] hostname
```

rlogin 建立一个从用户所在终端到远程主机 hostname 的远程登录会话。主机名被列在主机数据库中，该库可能包含在/etc/hosts 文件、网络信息服务(NIS)主机映射、Internet 域名服务器或它们的组合里。每台主机都有一个正式的名字(数据库记录的第一个名字)，还可以有一个或多个绰号。无论是正式主机名还是昵称都可以在 hostname 里指定。可以在本机的/etc/hosts.equiv 文件里保存一份可信主机名列表。

rm – 从目录删除文件

```
rm [-f] [-i] filename...
rm -r [-f] [-i] dirname...[filename...]
```

如果对文件有写权限，rm 就能删除一个或多个文件在目录中的对应记录。如果 filename 是一个符号链接，rm 会删除该链接，但链接指向的文件或目录不会被删除。用户要删除符号链接不必对它有写权限，只要具有对链接所在目录的写权限就行。

范例 A-46

```
1   rm file1 file2
2   rm -i *
3   rm -rf dir
```

说明

1. 从目录中删除文件 file1 和 file2。
2. 删除当前工作目录下的所有文件，但是事先询问是否删除。
3. 逐层删除 dir 下的所有文件和目录，忽略报错消息。

rmdir – 删除目录

```
rmdir [-p] [-s] dirname...
```

如果目录为空就删除该目录。如果制定了选项-p，同时删除父目录。

rsh – 启动一个远程 shell

```
rsh [ -n ] [ -l username ] hostname command
rsh hostname [ -n ] [ -l username ] command
```

rsh 连接到指定的主机 hostname，并执行指定的命令。rsh 把它的标准输入复制到远程命令，把远程命令的标准输出复制到它的标准输出，还把远程命令的标准错误输出复制到它的标准错误输出。中断、退出和终端信号也都被传给远程命令，rsh 通常在远程命令终止时结束。如果未指定命令，rsh 就用 rlogin 命令使您登录到远程主机。

范例 A-47

```
1   rsh bluebird ps -ef
2   rsh -l john owl ls; echo $PATH;cat .profile
```



说明

1. 连接到机器 bluebird 并显示该机上所有正在运行的进程。
2. 以用户 john 的身份进入远程主机 owl，执行 3 条命令。

ruptime – 显示本地主机的主机状态

```
ruptime [ -alrtu ]
```

ruptime 为局域网上的每台机器输出类似 uptime 的状态行。ruptime 依据网上的各主机每分钟广播一次的数据包构造出这个状态行。如果超过 5 分钟收不到某台机器的状态报告，ruptime 便显示该机器处于关机状态。ruptime 通常是按主机名排列状态行，但可以通过指定选项来改变该顺序。

rwho – 显示有哪些用户登录在本地主机上

```
rwho [ -a ]
```

rwho 命令生成类似于 who 的输出，但是它的查询对象是本地局域网上的所有机器。rwho 不能跨越网关，而且主机上必须运行有 rwho 守护进程，同时还须有目录 /var/spool/rwho。如果有 5 分钟未收到某台主机的报告，rwho 就假定它已关机了，不报告最后所知的登录在该机器上的用户。如果用户超过一分钟未向系统键入任何字符，rwho 就报告该用户处于空闲时间。如果用户有超过一个小时未向系统键入任何字符，除非指定了 -a 标志，否则，rwho 将在输出结果中略去该用户。

script – 创建一份终端会话的记录稿

```
script [ -a ] [ filename ]
```

script 生成一份记录稿，用于记录终端上打印出的所有内容。这份记录稿被保存为一个文件。如果未指定文件名，script 就将该记录稿保存为文件 typescript。script 在用户退出 shell 或按下 Ctrl+D 组合键时结束运行。

范例 A-48

```
1   script
2   script myfile
```

说明

1. 在新 shell 里开启 script 会话。终端上显示的全部内容都被保存在名为 typescript 的文件中。必须按^d 或退出 shell 才能结束 script 会话。

2. 在新 shell 里开启 script 会话。终端上显示的全部内容都被保存在文件 myfile 中。必须按^d 或退出 shell 才能结束 script 会话。

sed – 精简的编辑器

```
sed [-n] [-e script] [-f sfilename] [filename ...]
```

sed 把指定的文件 filename(默认为标准输入)复制到标准输出，文件的内容已被 sed 依

据命令或脚本编辑过。sed 不改变原文件。

范例 A-49

```
1   sed 's/Elizabeth/Lizzy/g' file
2   sed '/Dork/d' file
3   sed -n '15,20p' file
```

说明

1. 用 Lizzy 替换文件 file 中出现的所有 Elizabeth，并将结果显示在终端屏幕上。
2. 删除所有包含 Dork 的行并在屏幕上打印剩余的行。
3. 只打印第 15~20 行。

size – 以字节为单位显示对象文件每一节的大小

```
size [ -f ] [ -F ] [ -n ] [ -o ] [ -V ] [ -x ] filename...
```

size 命令为 ELF 或 COFF 目标文件里的每个被装入的会话生成以字节为单位的段或节的大小信息。size 打印出文本、数据和 bss(未初始化数据)段(或会话)的大小及它们的总和。

sleep – 挂起一定的秒数后执行

```
sleep time
```

sleep 挂起 time 指定的秒数后执行。它用来在一段时间后执行一个命令。

范例 A-50

```
1   (sleep 105; command)&
2   (In Script)
    while true
    do
        command
        sleep 60
    done
```

说明

1. 在 105 秒后，执行命令。提示符会立即出现。
2. 进入循环：执行命令并在再次进入循环前暂停一分钟。

sort – 排序和/或合并文件

```
sort [ -cmu ] [ -ooutput ] [ -T directory ] [ -ykmem ]
   [ -dfiMnr ] [ -btx ] [ +pos1 [ -pos2 ]] [ filename...]
```

sort 命令将所有命名文件里的行进行排序(ASCII 码)，并把结果写入标准输出。比较是基于从各输入行抽取的一个或多个排序键进行的。默认情况下，有一个排序键，即整个输入行，并且按机器排序序列里的字典排序法进行排列。

范例 A-51

```
1   sort filename
```



```
2  sort -u filename
3  sort -r filename
4  sort +1 -2 filename
5  sort -2n filename
6  sort -t: +2n -3 filename
7  sort -f filename
8  sort -b +1 filename
```

说明

1. 按字母顺序将行排序。
2. 排序输出重复的条目。
3. 逆序排序。
4. 排序从第一个字段(字段用空白符分隔并从 0 开始计算)开始，在第二个字段结束，而不是一直排序到行尾。
5. 按数值将第三个字段排序。
6. 按数值从第三个字段开始排序，并在第四个字段停止，把冒号当作字段分隔符(-t:)。
7. 排序合并大小写字母。
8. 排序从第一个字段开始，删除前导空白符。

spell – 查找拼写错误

```
spell [ -blvx ] [ -d hlist ] [ -s hstop ] [ +local_file ] [ filename]...
```

spell 从指定的文件中收集单词并在拼写列表中查找。既不在拼写列表中，也不是对列表中的词通过某种变形、加前缀或后缀而生成的单词将被打印到标准输出。如果没有指定文件名，则从标准输入中收集单词。

split– 将文件分片

```
split [ -n ] [ filename [ name ] ]
```

split 命令读取文件 filename 并将其分成大小为 n 行的一组小文件作为输出。第一个输出文件在文件名后追加 aa，然后以词典顺序追加直到 zz(最多 676 个文件)。name 的最大长度比文件系统所允许的最大文件名长度小 2 个字符。参见 statvfs。如果没有指定名称，则默认为使用 x(输出文件则为 xaa，xab 等)。

范例 A-52

```
1  split -500 filea
2  split -1000 fileb out
```

说明

1. 将文件 filea 分成 500 行大小的文件，依次命名为 xaa，xab，xac 并以此类推。
2. 将文件 fileb 分成 1000 行大小的文件，依次命名为 out.aa，out.ab 并以此类推。

strings – 在对象或二进制文件中查找可打印字符串

```
strings [ -a ] [ -o ] [ -number ] [ filename... ]
```

strings 命令在一个二进制文件中查找 ASCII 字符串。字符串是由以换行符或空白符结尾的 4 个或更多个打印字符序列。stings 可用于识别任意对象文件等其他文件。

范例 A-53

```
strings /bin/nawk | head -2
```

说明

打印出二进制可执行文件/bin/nawk 前两行中的所有 ASCII 文本。

stty – 为终端设置选项

```
stty [ -a ] [ -g ] [ modes ]
```

stty 为当前的标准输入设备设置特定的终端 I/O 选项。如果不带参数，stty 将报告这些特定选项的当前设置。

范例 A-54

```
1   stty erase <Press backspace key> or ^h
2   stty -echo; read secretword; stty echo
3   stty -a (AT&T) or stty -everything (BSD)
```

说明

1. 把退格键设置为删除。
2. 关闭回显功能，等待用户输入，重新打开回显功能。
3. 列出 stty 的所有可能选项。

su – 变成超级用户或其他用户

```
su [ - ] [ username [ arg ... ] ]
```

su 允许用户可以不必注销就成为另一个用户。用户名 username 的默认值是 root(超级用户)。要使用 su，必须提供正确的口令(或调用者已经是 root 了)。如果口令正确，su 便创建一个新的 shell 进程，该进程的真实和有效用户 ID、组 ID 和附加组列表按指定的用户名设置。这个新的 shell 将是由用户 username 的口令文件记录中的 shell 字段指定的那个 shell。如果未指定 shell，就使用 sh(Bourne shell)。如果要返回普通用户 ID 权限，可以键入 Ctrl-D 来退出新 shell。选项-可指定一个完整的登录。

sum – 计算文件的校验和

sync – 更新超级块并将改变过的块写回磁盘

tabs – 在终端上设置 tab 停止位

tail – 显示文件尾

```
tail +[-number [ lbc ] [ f ] [ filename ]
tail +[-number [ l ] [ rf ] [ filename ]
```



在 number 前加上一个加号，则 tail 将从文件第 number 行开始显示内容，显示内容可以是代码段、行或字符串。如果在 number 前加连字号，则 tail 将从文件尾开始计数。

范例 A-55

```
1   tail +50 filex
2   tail -20 filex
3   tail filex
```

说明

1. 从第 50 行开始显示 filex 文件的内容。
2. 显示 filex 文件的最后 20 行。
3. 显示 filex 文件的最后 10 行。

talk – 使您能够与其他用户交谈

```
talk username [ ttyname ]
```

talk 是一个可视化的通讯程序，它能从您的终端上复制文本行到另一用户的终端。

范例 A-56

```
talk joe@cowboys
```

说明

打开与机器 cowboys 上用户 joe 交谈的请求。

tar – 将文件保存到归档文件(通常是磁带设备)，或从归档文件中提取文件

```
tar [ - ] c|r|t|u|x [ bBefFhilmopvwX0134778 ] [ tarfile ]
[ blocksize ] [ exclude-file ] [ -I include-file ]
filename1 filename2 . . . -C directory filenameN ...
```

范例 A-57

```
1   tar cvf /dev/diskette
2   tar tvf /dev/fd0
3   tar xvf /dev/fd0
```

说明

1. 将当前工作目录下所有文件发送到设备/dev/diskette 的磁带上，并且打印出发送了哪些文件。
2. 显示磁带设备/dev/fd0 上内容的列表。
3. 提取磁带上的所有文件，并且打印出提取了哪些文件。

tee – 复制标准输出

```
tee [ -ai ] [ filename ]
```

tee 把标准输入复制到标准输出和一个或多个文件中，譬如 ls | tee outfile，这条命令将 ls 的输出结果写到屏幕和文件 outfile 中。

范例 A-58

```
date | tee nowfile
```

说明

将 date 命令的输出结果显示在屏幕上，同时保存到文件 nowfile 中。

telnet – 与远端主机通讯

范例 A-59

telnet necom.com

说明

打开与远端主机 netcom.com 的会话。

test – 计算表达式

test 计算表达式并返回一个退出状态以指示表达式为真(零)还是为假(非零)。该命令主要被 Bourne 和 Korn shell 用来进行字符串、数值和文件测试。而 C shell 中，大多数测试都是内置的。

范例 A-60

```
1   test 5 gt 6
2   echo $? ( Bourne and Korn shells)
   (输出为 1，表明测试结果不真)
```

说明

1. test 命令执行整数测试，检查 5 是否大于 6。

2. 变量$?保存了上一条命令的退出状态。如果退出状态非零，测试结果就不为真；如果返回的退出状态为 0，则测试结果为真。

time – 显示当前 shell 及其子进程的时间使用概况

timex – 为命令计时，报告进程数据和系统行为

```
timex [ -o ] [ -p [ -fhkmrt ] ] [ -s ] command
```

timex 将执行给定的命令，报告执行时消耗的总时间、用户时间和系统时间，以秒为单位。还可以选择列出或汇总该命令及其所有子进程的进程统计数据，报告命令执行期间所有的系统行为。timex 的输出结果被写到标准错误输出。

top – 显示顶层 CPU 进程

```
top [-] [d delay] [q] [c] [S] [s] [i]
```

top 命令可以实时查看 CPU 正在进行的活动并列出使用 CPU 最频繁的任务。



touch – 更新文件的访问时间和/或修改时间

```
touch [ -amc ] [ mmddhhmm [ yy ] ] filename...
```

touch 更新参数指定的各文件的访问时间和修改时间。如果指定的文件不存在，touch 就会创建它。如果未指定时间，touch 将使用当前时间。

范例 A-61

```
touch a b c
```

说明

创建 a、b 和 c 这 3 个文件。如果三者中某个已存在，则更新其修改时间戳。

tput – 初始化终端或查询 terminfo 数据库

```
tput [ -Ttype ] capname [ parms...]
tput [ -Ttype ] init
tput [ -Ttype ] reset
tput [ -Ttype ] longname
tput -S <<
```

tputs 使用 terminfo 数据库，它可以使 shell(参见 sh)获得与终端相关性能的值和信息，以初始化或重置终端，或者返回一个长长的所请求终端的类型名。

范例 A-62

```
1  tput longname
2  bold=`tput smso`
   unbold=`tput rmso`
   echo "${bold}Enter your id: ${offbold}\c"
```

说明

1. 显示从 terminfo 数据库中得到的终端的长名称。

2. 设置 shell 变量 bold，打开文本的突出显示。然后设置 shell 变量 unbold，返回正常的文本显示状态。"Enter your id:"这一行被突出显示为黑底白字，之后的文本则被正常显示。

tr – 转换字符

```
tr [ -cds ] [ string1 [ string2 ] ]
```

tr 把标准输入复制到标准输出，且替换或删除特定的字符。输入的字符如果在 string1 中，就会被映射成 string2 中的对应字符。可以把正斜杠加在八进制值的前面来代表字符的 ASCII 码。如果 string2(string2 中可以包含重复的字符)中的字符个数少于 string1 的，则 string1 中多出的那部分字符就不会被转换。可以用八进制值来代表字符，但要在前面添个反斜杠。

```
\11   制表符
\12   换行符
\042  单引号
\047  双引号
```

范例 A-63

```
1  tr 'A' 'B' < filex
2  tr '[A-Z]' [a-z]' < filex
3  tr -d ' ' < filex
4  tr -s '\11' '\11' < filex
5  tr -s ':' ' ' < filex
6  tr '\047' '\042'
```

说明

1. 把文件 filex 中的 A 都变成 B。
2. 将所有大写字母转换成小写。
3. 删除文件 filex 中的所有空格。
4. 将文件 filex 中多个连续的制表符替换(压缩)为单个制表符。
5. 将文件 filex 中多个连续的冒号压缩为单个空格。
6. 将来自标准输入的文本中的双引号替换为单引号。

true – 提供成功的退出状态

true 命令不做任何实际操作，只是返回真。即该命令总是返回退出状态 0，表示运行成功。Bourne 和 Korn shell 的 shell 程序用 true 命令来启动无限循环。

```
while true
do
  命令
done
```

tsort – 拓扑排序

```
/usr/ccs/bin/tsort [filename]
```

tsort 命令在标准输出上生成一个有序的项目列表，该列表与输入文件 filename 中给出的局部次序一致。如果未指定文件名，则从标准输入读取次序。输入由项目对组成(项目指非空字符串)，项目之间用空格分隔。由不同项目组成的项目对指明了项目间的次序。相同项目组成的项目对则表示其存在而非次序。

tty – 获取终端的名称

```
tty [ -l ] [ -s ]
```

tty 命令打印出用户终端的路径名。

umask – 设置创建文件时使用的权限模式掩码

```
umask [ ooo ]
```

用户的创建文件模式掩码被设置为 000。这 3 个八进制位分别代表属主、属组和其他用户的读、写、执行权限。创建文件时系统会将每个指定的数值从对应位上减去。如，掩码 022 将取消属组和其他用户的写权限(原来用模式 777 创建的文件，现在变成以模式 755



创建，原来用模式 666 创建的则变成用模式 644 创建)。如果省略 000，umask 就打印出掩码的当前值。umask 由 shell 来识别和执行。

范例 A-64

```
1   umask
2   umask 027
```

说明

1 .显示当前的文件权限掩码。

2. 目录权限 777 减去掩码 022 得到 750。文件属性 666 件去掩码 027 结果为 640。创建目录和文件时，其权限将设置为 umask 生成的值。

uname – 打印当前机器的名称

```
uname [ -amnprsv ]
uname [ -S system_name ]
```

uname 在标准输出上打印出当前系统的相关信息。如果未指定选项，uname 就打印出当前操作系统的名称。使用选项可以打印出 uname 或 sysinfo 返回的特定信息。

范例 A-65

```
1   uname -n
2   uname -a
```

说明

1. 打印主机的名称。

2. 打印主机的硬件名称、网络结点名、操作系统版本号、操作系统名称和操作系统版本，相当于同时使用-m、-n、-r、-s 和-v。

uncompress – 将之前用 compress 命令压缩的文件还原

```
uncompress [ -cFv ] [ file . . . ]
```

范例 A-66

```
uncompress file.Z
```

说明

将 file.Z 还原为原状态，即被压缩之前的状态。

uniq – 报告文件中的重复行

```
uniq [ [ -u ] [ -d ] [ -c ] [ +n ] [ -n ] ] [ input [ output ] ]
```

uniq 命令读取输入文件，并比较相邻行。正常情况下，重复行的第二个副本及之后的副本都被删除，剩余的内容则被写到标准输出。任何时候，输入和输出必须不同。

范例 A-67

```
1   uniq file1 file2
```

```
2  uniq -d -2 file3
```

说明

1. 删除文件 file1 中的重复行，将结果写入文件 file2。
2. 显示从第 3 个字段开始内容相同的重复行。

units – 将用标准度量单位的数值转换为其他度量单位

units 命令将标准度量单位的数值转换为其他度量单位的等值数。该命令以如下方式交互运行：

```
You have: inch
You want: cm
  * 2.540000e+00
  / 3.937008e-01
```

unpack – 展开用 pack 生成的文件

unpack 命令用于展开使用 pack 命令生成的文件。对于命令行中定义的所有文件名，unpack 将查找名为 name.z(如果 name 以.z 结尾，就查找 name)的文件。如果找到的文件是一个打包后文件，就将它替换为展开后的文件。新文件的名字去掉了后缀.z，其访问模式、访问时间和修改时间以及属主都和原来那个打包文件的一样。

uucp – 把文件复制到另一系统，UNIX 到 UNIX 的系统复制

```
uucp [ -c | -C ] [ -d | -f ] [ -ggrade ] [ -j ] [ -m ] [ -nuser ] [ -r ]
[ -sfile ] [ -xdebug_level ] source-file destination-file
```

uucp 命令将参数 source-file(源文件)指定的文件复制到参数 destination-file(目标文件)指定的位置。

uuencode, uudecode – 将二进制文件转换为 ASCII 文本文件，以便通过电子邮件发送它，后者将文件转换回原编码方式

```
uuencode [ source-file ] file-label
uudecode [ encoded-file ]
```

uuencode 将二进制文件转换为可通过邮件发送的 ASCII 编码形式。标签参数 file-label 指定了解码时使用的输出文件名。如果不指定文件，uuencode 将对标准输入进行编码。uudecode 读已编码的文件，剥离文件开头与末尾由邮件程序添加的所有行，然后还原出二进制数据，根据文件头部指定的文件名、模式和属主生成文件。编码后的文件是一个普通的 ASCII 文本文件，可以用任何文本编辑器来编辑，但是最好只修改头部的模式或文件标签，以免破坏解压出来的二进制数据。

范例 A-68

```
1  uuencode mybinfile decodedname > uumybinfile.tosend
2  uudecode uumybinfile.tosend
```



说明

1. 第一个参数 mybinfile 是需要编码的已有文件。第二个参数在文件被邮递后用来为解码生成的文件命名。uumybinfile.tosend 则是指通过邮件发送的文件。

2. 解码上一步用uuencode编码的文件，用给uuencode的第二个参数作为解码文件的名称。

wc – 统计行数、词数和字符数

```
wc [ -lwc ] [ filename ... ]
```

wc 命令用于计算文件中的行数、词数和字符数，如果未指定文件，则对标准输入进行统计。词指的是用空格、制表符或换行符分隔的字符串。

范例 A-69

```
1   wc filex
2   who | wc -l
3   wc -l filex
```

说明

1. 打印文件 filex 中行、词和字符的数目。
2. 用管道将 who 命令的输出发送到 wc，显示计算得到的行数。
3. 打印文件 filex 中行的数目。

what – 通过打印模式@(#)后的信息，从文件中提取出 SCCS 版本信息

```
what [ -s] filename
```

what 在 filename 指定的文件中查找模式@(#)(SCCS 的 get 命令用@(#)来代替关键字%Z%)，并且打印其后的内容直至遇到">、换行符、\或空(null)字符。

which – 定位命令并显示其路径或别名(UCB)

```
which [ filename ]
```

which 接受一组名称作为参数，并查找这些将名称看成命令时将要执行的文件。对于每个参数来说，如果它是别名，which 就扩展它，否则就在用户路径中搜索它。路径和别名都是从.cshrc 文件中取得。并且仅用到.cshrc 这一个文件。

whereis – 定位某个命令的二进制文件、源文件和手册页文件(UCB)

```
whereis [ -bmsu ] [ -BMS directory ... -f ] filename
```

who – 显示登录到系统上的用户

write – 给另一位用户写消息

```
write username [ ttyname ]
```

write 命令可以从当前用户的终端向另一用户的终端复制文本行。

xargs – 构造参数列表并执行命令

```
xargs [ flags ] [ command [ initial-arguments ] ]
```

xargs 命令允许将文件的内容传递给命令行，并动态地构造命令行。

范例 A-70

```
1  ls $1 | xargs -i -t mv $1/{} $2/{}
2  ls | xargs -p -l rm -rf
```

说明

1. 把目录$1 下的所有文件移到目录$2 下，并且在执行每条 mv 命令前先回显它。
2. 逐一提示(-p)用户要删除哪个文件，然后再删除它。

zcat – 把压缩文件解压到标准输出，相当于 uncompress –c

```
zcat [ file . . . ]
```

范例 A-71

```
zcat book.doc.Z | more
```

说明

解压 book.doc.Z，用管道将结果发送到 more。

zipinfo – 列出 ZIP 归档文件的详细信息

```
zipinfo [-12smlvhMtTz] file[.zip][file(s)...] [-x xfile(s) ...]
```

zipinfo 列出了通常用于 MS-DOS 系统的 ZIP 归档文件的技术细节。这些信息包括文件访问权限、加密状态、压缩类型、版本和操作系统或压缩程序的文件系统等。默认情况下(不带任何选项)以单行方式列出文档中所有文件条目，加上一些头部和尾部行以提供关于整个文档的统计信息。其格式结合了 UNIX 的 ls –l 和 unzip –v 的输出。

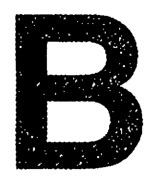
zmore –CRT 上的压缩文本文件过滤器

```
zmore [ name ... ]
```

zmore 是一个过滤器，用于在软拷贝(soft-copy)终端上全屏地检索压缩文件或普通文本文件。zmore 命令对使用 compress、pack 或 gzip 压缩的文件或未压缩的文件有效。如果文件不存在，zmore 就查找以.gz、z 或.Z 为后缀的同名文件。与 more 命令类似，它每次全屏地显示一个文件的内容。



appendix



各种 shell 的比较

对比 shell

特　　点	Bourne	C	TC	Korn	bash
别名	否	是	是	是	是
高级模式匹配	否	否	否	是	是
命令行编辑	否	否	是	是[*]	是
目录栈(pushd、popd)	否	是	是	否	是
文件名补全	否	是[*]	是	是	是
函数	是	否	否	是	是
历史	否	是	是	是	是
作业控制	否	是	是	是	是
键绑定	否	否	是	否	是
格式化提示符	否	否	是	否	是
拼写纠正	否	否	是[*]	否	是[+]

说明

[*]不是一个默认设置，必须由用户来设置。

[+]cdspell 是一个 shopt 选项。设置后，它会在用户使用 cd 命令时，纠正目录名中的拼写小错误。

tcsh 与 csh

TC shell(tcsh)是 Berkeley C shell(csh)的一个增强版本。这里列出了它的一些新特性：

- 增强的历史机制
- 用于编辑命令行的内置命令行编辑器(emacs 或 vi)



- 拼写纠正工具和用于拼写纠正和循环的特殊提示符
- 增强的和可编辑的单词补全功能，用于补全命令、文件名、变量、用户名等
- 创建和修改键绑定的功能
- 自动的、定期的、定时的事件(调度事件、专用别名、自动退出、终端锁定等)
- 新的内置命令(hup、ls –F、newgrp、printenv、which、where 等)
- 新的内置变量(gid、loginsh、oid、shlvl、tty、uid、version、HOST、REMOTESHOST、VENDOR、OSTYPE、MACHTYPE)
- 只读变量
- 更好的 bug 报告工具

bash 与 sh

Bourne Again(bash) shell 具有的下列特性，这些在传统的 Bourne shell(sh)中不具备。

- 格式化提示符
- 历史(csh 风格)
- 别名
- 用于编辑命令行的内置命令行编辑器(emacs 或 vi)
- 用 pushd 和 popd 对目录进行操作
- csh 风格的作业控制，可以用来暂停或运行后台作业、将后台作业带回前台等。例如，类似 bg、fg、Ctrl-Z 这样的命令
- 否定号、花括号、和参数扩展
- 定制键序列的键绑定
- 高级模式匹配
- 数组
- select 循环(来自 Korn shell)
- 许多新的内置命令

特性	C/TC	Bourne	bash	Korn
变量				
给局部变量赋值	set x = 5	x=5	x=5	x=5
指定变量属性			declare 或者 typeset	typeset
给环境变量赋值		NAME='Bob'; export NAME	export NAME='Bob'	export NAME='Bob'
只读变量				
访问变量	echo $NAME set var = net echo ${var}work network	echo $NAME var=net echo ${var}work network	echo $NAME var=net echo ${var}work network	echo $NAME 或 print $NAME var=net print ${var}work network

(续表)

特性	C/TC	Bourne	bash	Korn
只读变量(续)				
字符个数	echo $%var (仅 tcsh 适用)	N/A	${#var}	${#var}
专用变量				
进程的 PID	$$	$$	$$	$$
退出状态	$status, $?	$?	$?	$?
前一个后台作业	$! (tcsh only)	$!	$!	$!
数组				
给数组赋值	set x = (a b c)	N/A	y[0]='a';y[2]='b'; y[2]='c' fruit=(apples pears peaches plums)	y[0]='a'; y[1]='b'; y[2]='c' set –A fruit apples pears plums
访问数组元素	echo $x[1] $x[2]	N/A	echo ${y[0]} ${y[1]}	print ${y[0]} ${y[1]}
所有元素	echo $x or $x[*]	N/A	echo ${y[*]}, ${fruit[0]}	print ${y[*]}, ${fruit[0]}
第几个元素	echo $#x	N/A	echo $y{#[*]}	print ${#y[*]}
命令替换				
将命令的输出赋值给变量	set d = \`date\`	d=\`date\`	d=$(date)或 d=\`date\`	d=$(date)或 d=\`date\`
访问变量的值	echo $d echo $d[1], $d[2], ... echo $#d	echo $d	echo $d	print $d
命令行参数(位置参量)				
访问	$argv[1], $argv[2]或 $1, $2 ...	$1, $2 ... $9	$1, $2, ... ${10} ...	$1, $2, ... ${10} ...
设置位置参量	N/A	set a b c set \`date\` echo $1 $2 ...	set a b c set \`date\`或 set $(date) echo $1 $2 ...	set a b c set \`date\`或 set $(date) print $1 $2 ...
命令行的第几个参数	$#argv $# (tcsh)	$#	$#	$#
在$arg[number]中的第几个字符	$%1, $%2, (tcsh)	N/A	N/A	N/A

(续表)

特性	C/TC	Bourne	bash	Korn
用于文件名扩展的元字符				
匹配单个字符	?	?	?	?
匹配零个或多个字符	*	*	*	*
匹配字符集中的一个字符	[abc]	[abc]	[abc]	[abc]
匹配字符集中某个范围内的一个字符	[a–c]	[a–c]	[a–c]	[a–c]
匹配不在字符集中的一个字符	N/A (csh) [^abc] (tcsh)	[!abc]	[!abc]	[!abc]
? 匹配圆括号中某个模式的零或一次出现。竖杠代表一个或条件。例如，要么是2，要么是9。匹配abc21，abc91 或abc1			abc?(2|9)1	abc?(2|9)1
Filenames not matching a pattern	^pattern(tcsh)			
I/O 重定向和管道				
将命令的输出重定向到一个文件	cmd > file	cmd > file	cmd > file	cmd > file
将命令的输出重定向并追加到一个文件	cmd >> file	cmd >> file	cmd >> file	cmd >> file
将命令的输入是重定为来自一个文件	cmd < file	cmd < file	cmd < file	cmd < file
将命令的报错信息重定向到一个文件	(cmd > /dev/tty)>&errors	cmd 2>errors	cmd 2> file	cmd 2> errors
将输出和报错信息重定向到一个文件	cmd >& file	cmd > file 2>&1	cmd >& file 或 cmd &> file 或 cmd > file 2>&1	cmd > file 2>&1
将输出赋值并忽略 noclobber	cmd >| file	N/A	cmd >| file	cmd >| file

(续表)

特性	C/TC	Bourne	bash	Korn
I/O 重定向和管道(续)				
here 文档	cmd << EOF input EOF	cmd << EOF input EOF	cmd << EOF input EOF	cmd << EOF input EOF
将某条命令的输出经由管道发往另一条命令的输入	cmd | cmd	cmd | cmd	cmd | cmd	cmd | cmd
将输出和报错信息经由管道发往一条命令	cmd |& cmd	N/A	N/A	(参见协同进程)
协同进程	N/A	N/A	N/A	command |&
条件语句	cmd && cmd cmd || cmd	cmd && cmd cmd || cmd	cmd && cmd cmd || cmd	cmd && cmd cmd || cmd
从键盘读				
读取一行输入并保存到变量中	set var = $< set var = 'line'	read var read var1 var2...	read var read var1 var2... read read -p prompt read -a arrayname	read var read var1 var2... read read var?"Enter value"
算术				
执行计算	@ var = 5 + 1	var=`expr 5 + 1`	((var = 5 + 1)) let var=5+1	((var = 5 + 1)) let var=5+1
代字符号扩展				
代表用户的主目录	~username	N/A	~username	~username
代表主目录	~	N/A	~	~
代表当前工作目录	N/A	N/A	~+	~+
代表前一个工作目录	N/A	N/A	~-	~-
别名				
创建别名	alias m more	N/A	alias m=more	alias m=more
列出别名	alias		alias，alias -p	alias，alias –t
删除一个别名	unalias m	N/A	unalias m	unalias m
历史				
设置历史	set history = 25	N/A	automatic 或 HISTSIZE=25	automatic 或 HISTSIZE=25

(续表)

特性	C/TC	Bourne	bash	Korn
历史(续)				
显示有行号的历史清单	history		history，fc -l	history，fc –l
显示用行号指定的部分清单	history 5		history 5	history 5 10 history –5
重新执行一条命令	!! (last command) !5 (5th command) !v (last command starting with v)		!! (last command) !5 (5th command) !v (last command starting with v)	r (last command) r5 (5th command) r v (last command starting with v)
设置交互式编辑器	N/A (csh) bindkey -v 或 bindkey -e (tcsh 适用)	N/A	set -o vi set -o emacs	set -o vi set -o emacs
信号				
命令	onintr	trap	trap	trap
初始化文件				
在登录时被执行	.login	.profile	.bash_profile	.profile
每次调用 shell 时都执行	.cshrc	N/A	BASH_ENV=.bashrc(或其他文件名) (bash 2.x) ENV=.bashrc	ENV=.kshrc (或其他文件名)
函数				
定义一个函数	N/A	fun() { commands; }	function fun { commands; }	function fun { commands; }
调用一个函数	N/A	fun fun param1 param2 ...	fun fun param1 param2 ...	fun fun param1 param2 ...
编程结构				
if 条件语句	if (expression) then commands endif if { (command) } then commands endif	if [expression] then commands fi if command then commands fi	if [[string expression]] then commands fi if ((numeric expression)) then commands fi	if [[string expression]] then commands fi if ((numeric expression)) then commands fi

(续表)

特性	C/TC	Bourne	bash	Korn
编程结构(续)				
if/else 条件语句	if (expression) then commands else commands endif	if command then commands else ... fi	if command then commands else ... fi	if command then commands else ... fi
if/else/elseif 条件语句	if (expression) then commands else if (expression) then commands else commands endif	if command then commands elif command then commands else commands fi	if command then commands elif command then commands else commands fi	if command then commands elif command then commands else commands fi
goto	goto label ... label:	N/A	N/A	N/A
switch 和 case	switch ("$value") case pattern1: commands breaksw case pattern2: commands breaksw default: commands breaksw endsw	case "$value" in pattern1) commands ;; pattern2) commands ;; *) commands ;; esac	case "$value" in pattern1) commands ;; pattern2) commands ;; *) commands ;; esac	case "$value" in pattern1) commands ;; pattern2) commands ;; *) commands ;; esac
循环:				
while 循环	while (expression) commands end	while command do command done	while command do command done	while command do commands done
for/foreach	foreach var (wordlist) commands end	for var in wordlist do commands done	for var in wordlist do commands done	for var in wordlist do commands done



(续表)

特性	C/TC	Bourne	bash	Korn
循环(续):				
until	N/A	until command do commands done	until command do commands done	until command do commands done
repeat	repeat 3 "echo hello" hello hello hello	N/A	N/A	N/A
select	N/A	N/A	PS3="Please select a menu item" select var in wordlist do commands done	PS3="Please select a menu item" select var in wordlist do commands done

